Introduction to an emulation-based preservation strategy for software-based artworks

Klaus Rechert (University of Freiburg)
Patricia Falcao (Tate) and Tom Ensom (Kings College)
Commissioned by Tate

This research has been supported by and carried out within the framework of the PERICLES Project. PERICLES is a four-year Integrated Project funded by the European Union under its Seventh Framework Programme (ICT Call 9) and addresses work programme objective ICT-20114.3 Digital Preservation, under grant agreement no. 601138.

Abstract / Executive Summary

Emulation and virtualization offer a potential preservation strategy for software-based artworks by removing the artwork's dependencies on ephemeral physical computer hardware, which is prone to a short digital life-cycle. The implementation of an emulation strategy requires the introduction of new workflows, tools and novel risk-monitoring processes.

This report identifies three conceptual layers to a description of a software-based artwork and its environment:

1. Digital artefact description and configuration
2. Software runtime environment and configuration
3. Hardware environment

Using these layers, preservation requirements can be assessed separately for each conceptual layer, and risks identified through interdependencies between theses layers (particularly the technical interfaces between an artefact's software and hardware environment). An emulation-based preservation strategy focuses then, on the acquisition and maintenance of software environments, respectively their instances as virtual disk images and the monitoring of technical interfaces.

Focusing on an artwork's software environment rather than the artwork itself, highlights an interesting property of an emulation based strategy: much of the necessary work regarding emulation can be shared and standardized. While an emulation-based preservation strategy for a software-based artwork still requires a work-specific preservation plan to ensure the conceptual and technical verification process of its significant properties, work on the software and hardware environment layer (the primary concerns of an emulation-based strategy) is not work-specific. It therefore encourages the development of common practice as well as shared infrastructure and technical development. This report identifies important common tasks such as the identification and documentation of generalized emulated hardware environments and the migration of software environments between (emulated) hardware environments. It also highlights the need for systematic archiving of common software components that may form software environments. In many cases software environments can be shared among several artworks, pointing to a need for coordinated (and as a result, cost effective) efforts to keep these portable environments useable.

Emulation can also be used as a preservation tool to describe a software environment's hardware dependencies in an operating system and platform independent way. A software environment which was *successfully* migrated to an emulator setup has exactly the same hardware dependencies as the emulator provides. If an external dependency is accessible through an emulator setup, the nature of this dependency and the protocol communicating with it can be documented. If it is not accessible, a significant preservation risk is made explicit and can be addressed. If a software environment has been successfully rebuilt through emulation, a complete, verified and potentially machine readable description can be generated during this process.

Introduction

Keeping original born-digital artefacts accessible, usable and capable of being experienced in the long-term is one of the most challenging areas of digital preservation.[1],[2] As Matthews et al summarised:

> "Software has many characteristics which make its preservation substantially more challenging than that of many other types of digital object. Software is inherently complex, normally composed of a very large number of highly interdependent components and often forbiddingly opaque for people especially those who were not directly involved in its development. Software is also highly sensitive to its operating environment as a typical software artefact has a large number of other items upon which it depends including compilers, runtime environments, operating systems, documentation, and even hardware platform with its built in software stack. So preserving a piece of software may involve preserving much of the context as well." [3]

Emulation has been identified as a strategy that could be applied to the preservation of many born-digital artefacts. In as early as 1993 Doron Swade made the case for the "logical replication" of computer systems[4] and in 1999 Jeff Rotherberg[5] proposed it as a strategy for long-term preservation of digital objects.
In the context of this report we define born-digital artefact as the digital, software part of an artwork, excluding non-artwork specific software, for instance the operating system or a generic library. There is no uniform method to how these artefacts are built, and so they show great technical variety.
The preservation of digital art is concerned with maintaining a number of characteristics that must be maintained for us to say that the artwork had been preserved. These characteristics are determined on a case by case basis in dialogue with the artist. For some performing objects, including digital artworks, the way in which they were historically presented (or realized) may be valued. Therefore, it may be important to ensure that future realisations of the work maintain the characteristics of a previous performance, perhaps as far back as its

[1] NDIIP, Preserving*.exe,Toward a National Strategy for Software Preservation*, 2013 http://www.digitalpreservation.gov/multimedia/documents/PreservingEXE_report_final101813.pdf (online, last access 09/12/2016)

[2]*Digital Preservation Handbook*, 2nd Edition, http://handbook.dpconline.org/ Digital Preservation Coalition © 2015 licensed under the Open Government Licence v3.0.

[3] Matthews et al: Towards a Methodology for Software Preservation. In *Proceedings of iPres 2009, The 6th International Conference on Preservation of Digital Objects* pp. 132-140. https://escholarship.org/uc/item/8089m1v1#page-2 (online, last access Nov 24, 2016)

[4] Swade, Doron: The problems of Software Conservation. In *Computer Ressurection - The Bulletin of the Computer Conservation Society, Issue 7,* 1993. http://www.computerconservationsociety.org/resurrection/res07.htm#f (online, last access 09/12/2016)

[5] Rothenberg, Jeff *Avoiding Technological Quicksand: Finding a Viable Technical Foundation for Digital Preservation,* Council on Library and Information Resources, 1999 https://www.clir.org/pubs/reports/rothenberg/pub77.pdf (online, last access 09/12/2016)

first performance. Given this context, this report focuses on maintaining a software-based artwork’s computational performance when an emulation strategy is applied.
The availability and accessibility of digital artefacts’ is closely tied to the technical platform they were created on. These platforms, partly physical (hardware) and partly non-physical (software), are superseded by new platforms every five to ten years. Once the hardware parts are out of production, they inevitably disappear, rendering the software parts also inaccessible. To keep born-digital objects accessible, a promising approach is to focus on keeping the hardware platform alive by simulating the presence of the physical parts through virtual hardware. Virtualization and emulation provide the technical basis for a potential preservation strategy to keep digital performing objects accessible.

This report will focus on three issues:
Firstly, what types of artefacts are likely to be rendered successfully using today’s emulation technology? Secondly, what are the necessary steps to migrate an artefact from its original, physical environment to a virtual one? Finally, once an artefact has been migrated to a virtual hardware environment, what are the necessary actions to ensure long-term accessibility? Today’s emulators can also be considered as born-digital artefacts. An emulator which should protect an artefact from technical obsolescence can be exposed to a fast changing technical environment in a similar way to the artefact itself. Hence, a long-term strategy is required, relying on emulation technology on the one hand, and reducing reliance on a specific emulation platform (and its foreseeable technical obsolescence) on the other hand.
The report starts by defining the technical characterization of a digital artefact, and for the sake of clarity will divide the characterization between software and (virtual) hardware. It then explains the differences between emulation and virtualization, and their limitations in terms of access to peripherals - a key issue for the conservation of software-based artworks. Based on this characterization, possible options for an emulation-based strategy are offered, examining the creation of disk images and options to manage dependencies.

Technical characterization of a digital artefact

At first glance a digital artefact consists of a set of byte streams, e.g. binary files. Keeping these accessible in the long-term (i.e. being able to retrieve exactly the same byte stream as originally stored) poses some risks, but from today’s perspective this is a manageable task supported by established procedures and tools. In contrast, keeping a digital artefact’s features and experiential qualities accessible is a much harder task, since the artefact needs to be rendered, or performed (possibly involving interaction with the viewer) and depends on a suitable technical environment to do so.
To ensure that a digital artefact remains accessible and renderable in the long-term, it is essential that suitable software environments remain available. Emulation and virtualization of computer systems have been posited as potential long-term access and preservation strategies. To assess if emulation and/or virtualization are indeed suitable strategies, a set of high level questions are presented, intended to be considered either before, or at the time of, a work being accessioned into a collection. We will breakdown the assessment into more detailed questions in the following sections.

Question Set 1: Is emulation a potential preservation strategy?

Q: Does the artwork depend on a specific hardware and software environment to be rendered?

> Example: The artist supplied an executable file that runs in Windows XP. At acquisition the artist provided a computer with all the software installed.

Q: Are there alternative (less expensive) preservation strategies that can be considered?

> Example: If the source code is available and the artist is happy with it being changed, migrating the code to a more recent programming language or compiler suite may be a less costly option.

Q: Can the artwork be modified to improve compatibility with contemporary computer environments? Do you have the appropriate permissions to do that? Is it technically possible? Does it alter the experience of the artwork? How much would it cost?

> Examples:
>
> *Colors* (2009) is a software-based artwork by Cory Arcangel. It was developed using Objective C and C++, and the source code itself consists of a few hundred lines of code. The artist describes it as "An OSX app which plays a Quicktime movie one horizontal line of pixels at a time." Re-coding such functionality in a different language or environment would be relatively easy to achieve, and the artist has suggested this approach for preservation. On the other hand, trying to emulate this work would involve dealing with the constraints of running Mac OS X environments on non-Apple hardware. This is not technically impossible but breaches Apple's licence agreement for its Operating Systems[6].
>
> At the other end of the spectrum, another work in Tate's collection is *Sow Farm (near Libbey, Oklahoma) 2009* by John Gerrard. This work was developed using 3D asset production software and the Quest3D visualisation/simulation software. The compiled software application consists of an executable file and supporting configuration files. Due to the complex software tool chain used in its creation and the one-way transformation from development environment to executable software, migrating this software is likely to be very resource intensive, and having the software used to create the work would be crucial.

[6] See licence agreement for Mac OSX1011 (El Capitan) here http://images.apple.com/legal/sla/docs/OSX1011.pdf, (online, last access Nov 24, 2016) which states "Apple Boot ROM code and firmware is provided only for use on Apple-branded hardware and you may not copy, modify or redistribute the Apple Boot ROM code or firmware, or any portions thereof."

There are two main strategies that can be used for long-term access and preservation: ***migration*** *- adaption of the artwork to the current environment, e.g. by re-creation, re-compiling or modification; and* ***emulation/virtualization*** *of its technical rendering environment. Preference for one or the other will depend on what elements of an artwork are available, the requirements of the artist and custodians of the work, and how resource-intensive it would be to use one in comparison to the other (Question Set 1). This latter point applies particularly in the long-term, as both methods have to be re-applied due to foreseeable technological change. In many cases both approaches are possible, and ideally both options should be planned for.*
Legal issues may also need to be taken into account, specifically for Mac OS-based systems, which rely on ROMs with unclear licensing.

Recommendations: If possible, plan for both migration and emulation strategies when obtaining information about, and acquiring the software elements of, an artwork. Sometimes only one of the strategies may be applicable, and this should be ascertained as soon as possible.

Software Layer

Many digital artefacts are not self-contained. They do not only require hardware, but also additional software to be rendered. Therefore, one part of an artefact's technical environment represents a software runtime – the *software (rendering) environment*. A minimal software environment is typically an installed and configured operating system, but in most real-world scenarios a more complex software environment with additional software installed and configured is required.

Question Set 2: Assessing the digital artefact (artefact composition)

Q: Is a binary / executable of the artwork's software available?

- Can the artwork's software be reinstalled? Is an automated/guided installation process available? Is the software installation process documented?
- Is the software environment documented, in particular are the artefacts technical dependencies documented? What are the (minimal) technical requirements to run the executable (operating system, software, libraries, etc.) ?

Q: Is the artwork's source code available?

- Can the artwork's software be re-built (e.g. from source)? One may need the necessary tools or 'tool chain' used to build and package all the elements (e.g. compiler used, development environment, the tool-chain's configuration etc.).
- Are there appropriate replacements / options to modify the code if necessary?

Q: Is there a technical operation manual? Are specific configuration steps necessary? How is the software correctly executed (e.g. is a calibration process necessary)? Is there detailed documentation about the artworks expected behaviour?

If the software environment is not known, an artefact may be preserved using its original environment as a reference (e.g. the artist's original computer). This reference environment can be examined in order to determine the artwork's software requirements, and the artwork can be isolated from its original environment and rendered in another technically compatible environment.

Question Set 2: assessing the digital artefact (computer system)

Q: Is the original computer setup available for examination and as reference?

Q: Can the artwork be isolated and extracted from the running machine? e.g. unique software components can be distinguished from generic components, with the goal of using a generic image / rendering environment?

If the configuration of a software environment is known, the software environment can be rebuilt if necessary. Either using documentation derived from a reference setup or systematically determined software dependencies (e.g. using tools for analysing an artefact's technical format), a suitable software rendering environment can be re-built by installing and configuring an artefact's software dependencies. The installation procedure can be performed manually or in an automated way (e.g. if a machine readable description of the software environment is available). The process of rebuilding an artefact's rendering environment can also be used to create verified and machine readable technical metadata and in doing so support long-term preservation and access. Machine readable installation information (e.g. what type of software is installed), and more importantly the environment's configuration, can be captured during the structured software environment rebuilding workflow.

Question Set 3: software environment acquisition options (rebuilding)

Q: Are the necessary software components available for (re-)installation? E.g. configuration files, compilers, and all other tools and configuration files required to reproduce the build process)

Q: Is there a strategy to preserve software (including managing licenses)?

Q: Are preserved software components complete (including their own dependencies) and useable (e.g. installable)?

Q: Is the configuration of individual software components known (e.g. required fonts, language settings, network / proxy configuration etc.)?

If the approach of re-building the software environment is not an option, or if it fails (e.g. all software dependencies have been successfully identified but configuration details of some software components are missing or unknown) imaging the original computer's disks for use with emulators remains an option.

Question Set 3: software environment acquisition options (generalising)

Q: If the artwork is available as a complete computer system (rather than as an independent executable file or as an online work distributed among multiple servers), is imaging the computer's hard disk(s) an option?

Q: Is the installation still functional, i.e. is the original (e.g. the artist's) computer still available and accessible as a reference implementation?

Q: Is preservation of a disk image a suitable or sufficient strategy? Relying on a disk image means that this image most likely requires modifications to run in a different technical environment. Depending on available knowledge about the image's source and internal configuration, it may be difficult to assess what type of modifications are necessary and how they should be carried out.

The first step for a successful emulation strategy is knowing what information and resources are available to support this process (Question Set 2). Having source code available (enabling the recreation of the artefact on a current machine) allows for a higher technical abstraction level and may also open the door to alternative strategies (e.g. migration). If source code is not available, creating an emulated version of the work may be the only suitable choice. How this emulated version is then created will depend on whether the software can be re-installed and all the required software dependencies are available and operational (Question Set 3). The simplest but most limited option is to rely solely on disk images of the original computer.

Recommendation: Collect information to support emulation in a timely manner. In particular, obtain source code and build instructions (if available), and install the artefact as soon as possible in an emulated / virtualized environment - while an original reference installation and/or the artefact's creator are available.

Interfaces between Hardware and Software

Operating systems (OS) play an important role in software environments, as they typically provide a hardware abstraction layer, for instance, any application is able to connect to the internet without knowing technical details about the hardware used. This abstraction is usually achieved through *technical interfaces* - the so called hardware abstraction layer. The technical interfaces between an operating system and hardware have two sides: the top-side (OS-side) unifies usage of different hardware components (e.g. sound card, network card, graphic cards etc.) and the bottom part (hardware-side) operates the hardware in (vendor-) specific ways. The connection between top- and bottom interfaces are implemented as hardware drivers (sometimes as BIOS/ROMs[7] extension). A popular example of a hardware abstraction layer is Microsoft's Network Driver Interface Specification (NDIS)[8], which defines a set of network functions that abstract the complexity of the underlying network hardware. A program requiring network functionality could then be programmed using these abstract functions, while a specific hardware driver (the Media Access Control (MAC) layer) is in charge of translating these functions to hardware actionable instructions. In this way, the network card can be exchanged for another model without the need to rewrite applications, simply by installing a new (NDIS) hardware driver.

Through the use of OS hardware abstraction, dependencies on physical hardware components are usually unnecessary. Software artefacts typically pose only abstract hardware dependencies (e.g. the minimal screen resolution, sound and network support etc.) This approach has greatly simplified software development and improved the compatibility of software with a wide spectrum of different computer setups. Most artefacts and software dependencies typically use operating systems to interact with hardware.

Some digital artefacts, however, have no software dependencies and are able to interact with hardware components directly. However, these cases are rather rare (at least for typical computer setups) and usually associated with specific hardware - for example, game

[7] Apple Macintosh Computers are a good example for relying on ROMs

[8] Network Driver Interface Specification, https://en.wikipedia.org/wiki/Network_Driver_Interface_Specification, (online, last access Nov 11, 2016)

consoles, arcade machines, robots or similar special purpose machinery. There are also some rare cases where an operating system is used but the artefact also relies on direct access to a specific hardware resource.
In general, for artefacts or software environments interacting directly with hardware, there is a higher risk of an emulation strategy failing, in particular if they rely on custom built or modified hardware components. Even if they rely on widely used hardware, not every feature (and possible quirk) of real physical hardware components may be emulated accurately. Furthermore, emulators may have bugs or behave differently compared to the original systems. In contrast, artefacts relying on operating systems to interact with emulated hardware are more likely to be successfully re-enacted using emulation, as emulator implementations usually try to cover the feature-set of popular hardware (and drivers) and/or the behaviour of a popular operating system.

Question set 3: software environment acquisition options (identify an emulator)

Now that the hardware and software environment has been identified, the following questions serve to guide the selection of an emulator; maximising the chances of successfully running of the software.

> Q: Is there an emulator (or virtual machine setup) available supporting the identified software stack, in particular its operating system?
>
> Q: In case of artefacts running directly on hardware: is there an emulator for that particular hardware platform?
>
> Q: Are there specific requirements regarding the emulators performance (e.g. input latency, video/audio sync, throughput, compute performance)?
>
> Q: Is the list of drivers / technical interfaces known?
>
>> Example: If the list of drivers is known (and potentially also the hardware used with these drivers), an a-priori assessment of a hardware migration becomes possible. The hardware equipment of an (emulated) computer system can be compared against the hardware in use by the OS in the physical computer.
>
> Q: How does software communicate with hardware? Are standard protocols or interfaces used? (e.g. OpenGL, DirectX, NDIS)
>
> Q: Are any specific hardware components used with the original artwork? (e.g. Graphics Card or USB video camera)
>
> Q: Which hardware components are active (in use) and what kind of drivers are used?
>
> Q: Is the hardware essential for the artwork and does it have some specific functionality that is required?
>
>> For instance, in *Subtitled Public*, by Rafael Lozano-Hemmer the software analyses images received from a video camera to identify visitors in a space, the live video input is essential to the work, even if the video cameras are not)
>
> Q: Is the (vital) hardware a component of the executing machine (e.g. built in)?
>
> Q: Is the hardware used essential to the artwork’s performance?
>
>> If yes, what are the specific characteristics or features used? Are these abstract feature sets (such as DirectX / OpenGL for 3D rendering) or specific hardware features (e.g. those provided by a specific device)? If the former, can the hardware be substituted for a different model?

Q: Is the hardware component 'visible' (i.e. directly used by the digital artefact), bypassing drivers / abstraction layers or using specific driver features?
Q: Does the hardware bypass any part of the OS?

> Example: An artwork using a high-end 3D graphics card: The card may be used just because of the faster processing speed, but the digital artefact may also be making use of specific 3D features through an OS abstraction layer (e.g. DirectX / OpenGL / etc). Sometimes the digital artefact may even be making use of specific features of the graphics card itself.

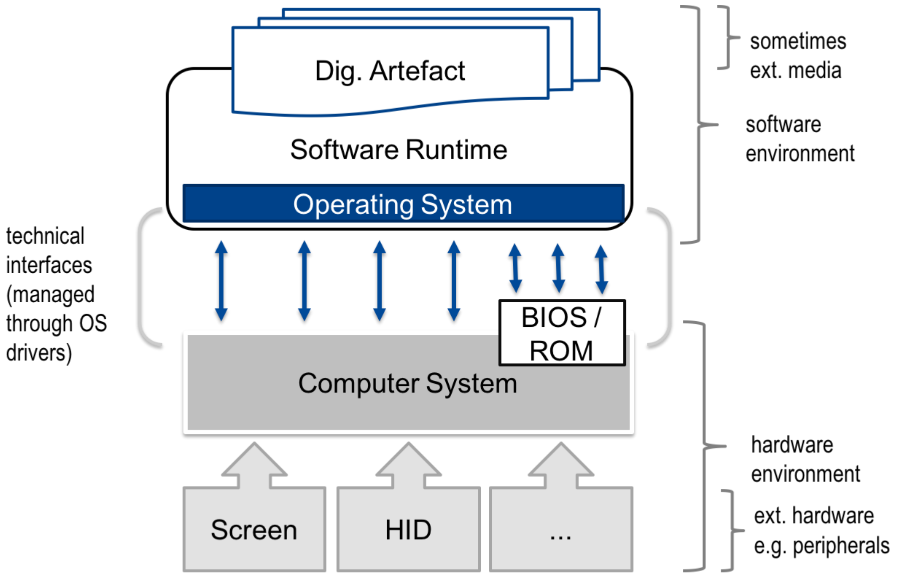


Figure 1: Rendering environment of a digital artefact

Technical interfaces form an abstraction layer between the software environment (typically the operating system) and hardware, unifying the usage of different hardware components and, through hardware drivers, operating the hardware in vendor-specific ways. This is the layer addressed when preparing an existing computer setup for emulation.
One of the key requirements for a successful (long-term) emulation strategy is to prevent direct usage of specific hardware by an application, and use as small as possible a variety of emulated hardware in its place.

Recommendation: Identify hardware components or interfaces that may not be supported by emulators and determine equivalent (emulated) substitutes. E.g. a computer may use a video card that has itself not been emulated, but when creating a disk image a different emulated video card can be used to replace the original hardware card.

Hardware Layer

The hardware layer connects the digital artefact (and its software runtime) with the physical world. Hardware components, as considered in this report, can be classified into two classes: a ‘machine’ with built-in hardware (e.g. a computer, phone, tablet etc.), and hardware components connected to this machine.

For the purpose of this report the hardware is only relevant in so far as it influences the options for emulation or virtualization. The focus of this section then, is on the characteristics to document in hardware when considering its use for emulation or virtualization purposes.

When an artwork is acquired it is important to analyse and describe the hardware used by a digital artefact, as this will help to define the technical environment required for that digital artefact to be rendered.

The level of detail needed to describe the hardware depends on the characteristics of the software environment where the digital artefact is run. Preservation goals will also influence the information needed. The machine’s architecture and all its built-in components (e.g. sound card, storage controller, network card, graphics card) should be documented as this information is needed to guide the search for a suitable emulator or virtualizer. In cases with no detailed information about the software environment (e.g. operating system used) is available, hardware information is also useful -providing insights on potential software environment configuration (e.g. for older ROM-based computer systems[9]).

Hardware analysis is essential to understanding options and difficulty in emulating or virtualizing a digital artefact. In addition to understanding the computer system, it is just as important to define the software environment in which the artefact is running. Combining these two sources of information highlights the technical interfaces involved, and ensures an understanding of future risks for an emulated version.

Recommendation: Analyse how the digital artefact and/or its software dependencies utilize hardware and which (if any) specific features of the hardware are vital.

Virtual Hardware

Any computational (binary) operation can be implemented in hardware (i.e. hard-wiring operations for a specific purpose as a machine or machine component) or in software (i.e. translating a set of complex logic operations by compiling source code into instructions for a generic machine - e.g. any generic CPU). Both implementation options are in theory equivalent, however operations implemented in hardware are usually by magnitudes faster compared to a pure software implementation. This equivalence allows, however, the replication of any outdated hardware component in software and the exposing of its functionality using contemporary hardware. Hence, the ‘Computer System’ block, depicted in Fig. 1 can be implemented either as physical or virtual hardware.

There are currently two generic technical options to replace outdated hardware: virtualisation and emulation. These two technologies are not mutually exclusive, and in fact share many concepts.

[9] A ROM was part of the computer system’s mainboard containing basic operating system code (firmware).

Virtualisation

Virtualisation is a concept and a technical tool to abstract (“virtualize”) hardware resources, such that a so called *virtual machine* (VM) (also called ***guest***) is not interacting with the physical hardware directly. Instead a *hypervisor* (also called ***host***) provides a virtual hardware interface for guests, usually providing access to a unified set of hardware, regardless of the machine’s hardware configuration the host is running on. A virtual machine is still able to utilize performance advantages of real hardware, in particular (but not restricted to) using the host’s CPU. The hypervisor is in charge of enforcing rules as to how a virtual machine is able to use the host’s CPU (i.e. restricting access to specific, sensitive instructions - for instance, preventing a VM accessing the host’s memory), such that a VM is unable to takeover the physical machine or interfere with other VMs running on the same host. Modern computer architectures have built-in hardware features (e.g. dedicated CPU and memory-management features) to support virtualization, implementing parts of the hypervisor in hardware and thus reducing the virtualization overhead as well as the complexity of the host system (software hypervisor).

Hardware components available to guest VMs are either fully emulated or, to improve performance by eliminating most of the virtualization penalty/overhead, *para-virtualized*. In order to para-virtualize hardware a thin software layer is installed within the virtual machine (typically as a driver), to allow access to a host machine’s hardware component. Para-virtualized hardware offers (almost) direct and efficient access to the host’s hardware, typically (but not restricted to) network cards and storage controllers (e.g. disk access). When using para-virtualized hardware, a driver specific to the virtualisation system needs to be installed within the guest system. In contrast, when using fully emulated hardware components, the guest system is able to use the same driver code as if it were using a physical hardware component.

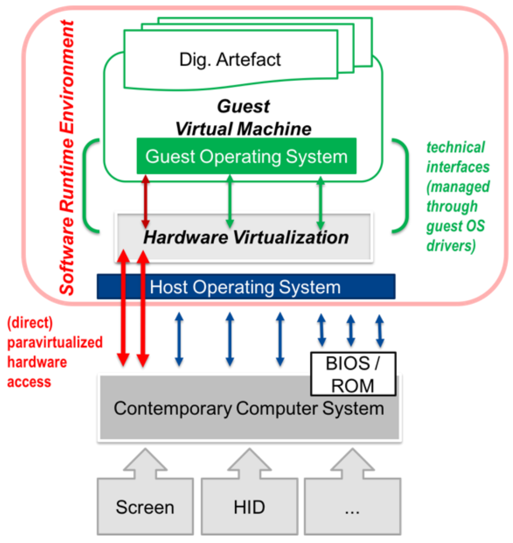
(Commercial) Examples of virtualization systems: VMware, VirtualBox, qemu-kvm.

Emulation

An emulator (usually) implements a specific outdated computer system, primarily a CPU architecture, interpreting the outdated machine’s instructions and translating these to equivalent instructions of the current host system. This however, is not sufficient to run or execute even the most simplistic applications. There is additional hardware emulation required to attach storage devices (e.g. a bus to a floppy or hard-drive controller), a memory management unit (MMU), video and audio output, network interfaces, and interfaces for interaction with users. In contrast to virtualization, emulation implements a complete computer system in software. Therefore, an emulated system is independent of the current computer architecture, e.g. we are able to run a Motorola 68k emulator (e.g. to boot MacOS System 7) on a current Intel-based Windows PC.

Popular emulator examples: qemu, Sheepshaver (Mac PPC), BasiliskII (Mac M68K)

Choosing between Virtualisation and Emulation


Software Runtime Environment
Dig. Artefact
Guest
Virtual Machine
Guest Operating System
technical interfaces (managed through guest OS drivers)
Hardware Virtualization
Host Operating System
(direct) paravirtualized hardware access
BIOS / ROM
Contemporary Computer System
Screen
HID
...

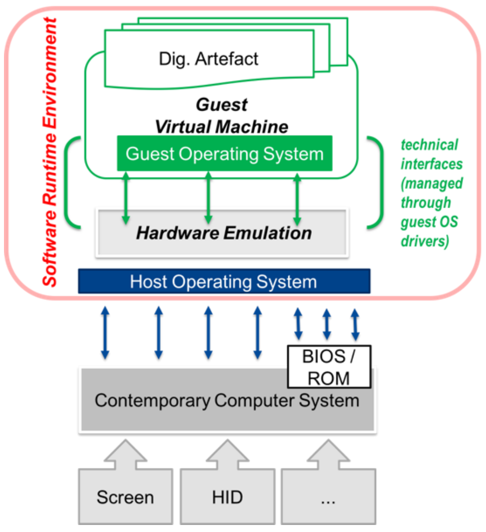


Figure 2: Schematics of a virtualized computer system and its reliance on hardware interfaces of the host system.

Figure 3: Schematics of an emulated computer system.

The main difference between emulation and virtualization is the reliance on contemporary hardware (or the lack thereof). Virtualization relies on and utilizes real hardware for performance reasons as it offers typically (almost) native execution speed, but has restricted platform support. By definition, virtualizers (such as VirtualBox and VMWare), are only able to run Intel-based x86 VMs, whereas emulators cover almost any technical platform, in particular obsolete ones. Furthermore, the close ties to today's computer platforms restrict a virtualized machine's longevity, particularly if the virtual machine relies on contemporary (para-virtualized) hardware components. To support para-virtualized hardware, the VM (and the virtualization technology) not only rely on VM-specific drivers installed in guest systems, but these drivers also expect appropriate support from the host OS, typically as a host-OS kernel extension to support interaction with the host's hardware directly. Any major OS-kernel upgrade (e.g. changes to internal kernel interfaces) requires an upgrade of the virtualizer too, and therefore the longevity of the virtual machine guest depends also on the vendor or community supporting current operating systems.
As all hardware components (and respectively their low-level interfaces to be used by drivers) of a computer system have to be re-implemented in software, and the availability of drivers for old operating systems is crucial, only a small set of emulated hardware components are provided as virtual / emulated hardware. Typically, emulator developers have focused on hardware in widespread use and with good driver support, e.g. Soundblaster 16/32 for soundcards and Intel's E10/100/1000 for network cards. In practice there is a significant overlap between virtualizers and emulators, with both supporting a similar set of emulated hardware components, a useful property for a mixed preservation strategy. Hence, for digital preservation and related tasks, one should avoid extensions or drivers specific to a certain virtualizer or emulator. If possible, drivers originally published by hardware vendors should be used, since using the original driver also verifies (at least partly)

correctness and completeness in the emulated hardware. Furthermore, by using emulated standard hardware and their drivers, both the effort and risk of migrating a virtualized system to an emulated one is reduced.

Contemporary emulators require a specific contemporary host system, which is to say that emulators are normal software components with specific requirements regarding their software and (abstract) hardware environment. However, the guest systems running on emulated hardware are usually not specifically configured for a certain emulator (e.g. using original hardware drivers). Hence, migrating an emulated guest to a new emulator of the same hardware, will require little or ideally no adaptation of the system.

To summarize, using virtualization technology can be a useful addition to a general emulation strategy, in particular if performance matters. Even though a single virtualisation solution can not be considered a long-term solution, if carefully configured (e.g. avoiding para-virtualized drivers) the effort required to migrate a system to a new virtualizer or emulator is lowered. Furthermore, having two similar systems at hand (e.g. VirtualBox for virtualization and QEMU for emulation) offers the option to pursue a two-track strategy, and in particular allows the practice of system migration between two virtual hardware stacks.

Computational operations can be implemented in both hardware or software. This equivalence allows the replication of any outdated hardware component in software, exposing its functionality to software in a contemporary hardware environment. Two different techniques can be used to achieve this:

- *Emulation implements a complete computer system in software and as a consequence is independent of the host computer architecture, e.g. running a Motorola 68k emulator to boot MacOS System 7*
- *Virtualisation creates a so called virtual machine (VM) (also called guest), which is able to interact with the physical hardware indirectly. Instead of interacting directly, a hypervisor (also called host) provides a virtual hardware interface for guests and (usually) a unified set of hardware, regardless of the hardware configuration the host is running on. This means that for instance, a server can host multiple VMs with different operating systems, as long as these use the same architecture as the underlying hardware.*

Recommendation: Virtualisation technology is a viable intermediate solution for current, process-intensive software and/or recent operating systems. As a long-term solution, full hardware emulation is usually a more appropriate option, as it is more likely to run a fully emulated system unmodified on a next-generation emulator - particularly if the emulator developer community agrees on a standard hardware configuration.

Characterization of External Dependencies

A digital artefact is often composed of different digital objects which, to be rendered, require more than an isolated technical environment (consisting of data and/or extracted disk images) and a computer system or installation. For this reason, external dependencies need to be determined, characterized and documented.

Based on the aforementioned structural view and discussion, a (minimal) technical description of a digital artefact can be divided into three conceptual layers:

1. ***Artefact Description & Configuration***: This layer conceptually captures the technical description of the object, in particular the technical properties of the artefact itself (e.g. a reference to its bitstream representation, technical information about file formats and their properties, hardware- and software dependencies, source code, etc.) and its specific (artefact-specific) configurations and settings.
2. ***Software Environment & Configuration:*** This layer conceptually captures all installed software components and applications, including the operating system (if present). Furthermore, it may capture the configuration and settings of individual software components and the operating system.
3. ***Hardware Environment:*** This layer conceptually captures the hardware components used by layers 1 and 2.

The three logical layers together describe the technical environment and characteristics of a digital artefact. In each layer, individual components may depend on functionality or data not available within the local setup (*direct* external dependencies). Additionally, there may be *indirect* external dependencies. For instance, a direct software dependency originating from the artefact layer may itself have external dependencies.

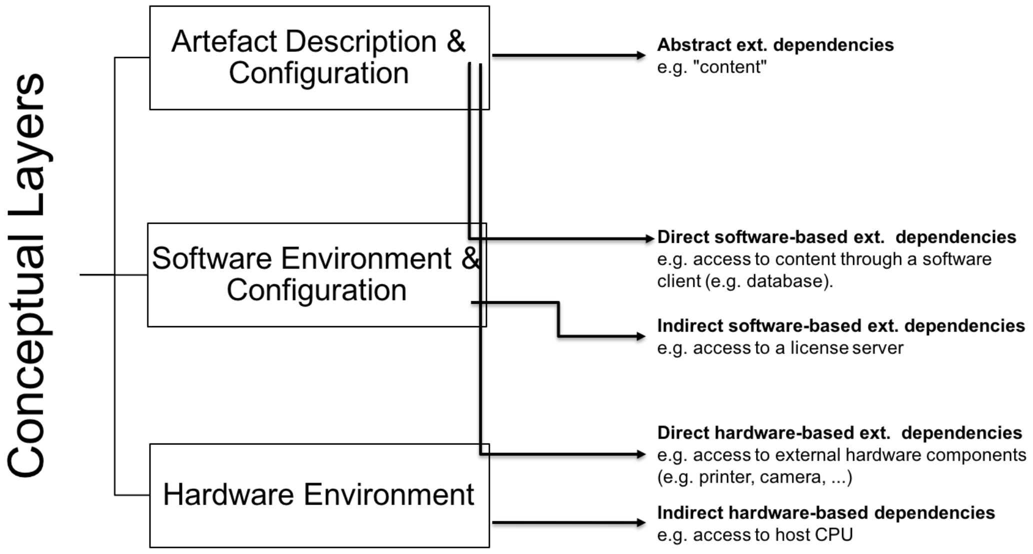

Conceptual Layers
Artefact Description & Configuration
Software Environment & Configuration
Hardware Environment
Abstract ext. dependencies
e.g. "content"
Direct software-based ext. dependencies
e.g. access to content through a software client (e.g. database).
Indirect software-based ext. dependencies
e.g. access to a license server
Direct hardware-based ext. dependencies
e.g. access to external hardware components (e.g. printer, camera, ...)
Indirect hardware-based dependencies
e.g. access to host CPU

Figure 4: Characterization of external dependencies derived from conceptual layers

From the three conceptual layers we can derive the following five types of external dependency:

1. **Abstract external dependency**: Abstract dependencies are posed directly by the digital artefact. These dependencies are abstract in that they do not rely explicitly on a specific software or hardware component. For instance, an artefact's performance might depend on access to some kind of data source stored at an external site, but does not rely on specific software to retrieve or modify the data (e.g. the data is directly accessible through the local file system).

2. **Direct software-based external dependency**: To access external data and/or functionality the artefact requires additional software. For instance, a specific client software is required to connect to a remote database and retrieve data.
3. **Indirect software-based external dependency:** This type of external dependency is not posed by the artefact directly but by another of the artefact's software dependencies. It is therefore called an indirect software dependency. For instance, database client software might require access to a license server to function.
4. **Direct hardware-based external dependency:** The digital artefact requires access to external hardware, such as direct access to a specific printer.
5. **Indirect hardware-based external dependency:** The software environment of the digital artefact requires access to specific hardware. For instance, a virtualisation system which requires access to a specific CPU type to function, or a software component requires access to a hardware licence dongle to function.

External (Connected) Hardware

An important subset of external dependencies are external (connected) hardware components, which can be seen as direct or indirect hardware-based dependencies. A general characterization of external hardware is beyond the scope of this report.

This section will focus on the characterization of the communication between a virtualized or emulated machine and external hardware, as well as data protocols used, (i.e. how information is exchanged between software and external hardware). This is the essential information needed when considering the use of emulators to run an artefact.

To (re-)connect external hardware components a physical machine is required. In the case of an emulated or virtualized computer system, the host system is able to connect and interact with external hardware components such as human interface devices (HID) (e.g. mouse and keyboard), printers or other peripherals.

The first issue to be solved in connecting external hardware to an emulated or virtualized computer system, is establishing a physical connection between the system hosting the virtual guest and the external hardware component. For this to happen the current computer system needs a suitable connector or a passive (if electrically compatible) or active (e.g. analogue-to-digital converter) adapter, to provide a compatible connection.

Second, the host operating system needs to provide a software interface for applications to communicate with external hardware. For example, a *COM port*, the Windows software-side representation of a serial connection used for generic peripheral devices such as printers. If external hardware is accessible through the host OS's interfaces, an emulator (acting as a normal software application) is then able to use this external hardware.

Finally, the emulator needs to provide a virtual hardware interface connected to the host software interface, visible to and usable by the guest OS.

Through all these layers' integrity of the data protocols, used to communicate between software within the emulated environment and external hardware, needs to be maintained.

Similarly to the previously discussed built-in hardware components, judging the relevant and controllable risks posed by an external component on the complete installation should be focused on its technical interfaces.[10] Fortunately, the number and type of external technical

[10] The preservation of the physical device as well as any conceptual issues of external components are not considered in this report.

interfaces is low. Their types are standardized and mostly use general purpose technologies (such as USB, serial/parallel port etc.), providing the general computer architecture does not change significantly.
Some older external components and interfaces aren't supported by emulators anymore, mostly for practical reasons, such as host systems providing no appropriate connectors (e.g. a mobile phone or tablet being used as an emulator host has limited connector options). In these cases, usually an emulated or simulated option is provided (e.g. a GUI console gamepad emulation on a touchscreen device).

Question Set 4: external dependencies

Q: Does the artwork depend on a specific network environment (servers to be accessible, NAS, etc.)- In many cases external, but local data sources can be integrated or consolidated.
Q: Does the artwork depend on external data sources (e.g. is web-based)?

Q: Is there any external hardware connected?
Q: Is it generic hardware (e.g. peripherals – mouse, keyboard) or specific (e.g. a particular type of video camera) or unique (specifically made for purpose).
Q: What technical interfaces and protocols is the hardware using (e.g. serial)
Q: Are these interfaces still in use (i.e. can the hardware be connected to the host computer) and is their software support for the host system supporting the protocol used
Q: Is timing critical , i.e. are there hard deadlines for the OS to react on events?
Q: Is software emulation or simulation of external hardware an option?

Three different but related layers must be considered when describing a digital artefact: artefact description and configuration, software environment and configuration and hardware environment.
An analysis of these layers identifies five types of external dependency: Abstract dependency, direct software dependency, direct hardware dependency, indirect software dependency and indirect hardware dependency.
External direct hardware dependencies are particularly relevant for software-based artworks, and so specific solutions may need to be found.

Recommendation: Identify external dependencies and the conceptual layer these dependencies originate from. Identify and document technical interfaces needed to utilize external dependencies, as well as their relative risks for the performance of the artefact.

An Emulation-based Preservation Strategy

In the previous section three conceptual layers describing the technical characteristics of a digital artefact were identified. The structural separation of hardware, software environment and digital artefact facilitates the evaluation of preservation risk factors and strategies without considering the specificities of all layers. For instance, a computer's physical hard disk can be removed from one computer system and built into another computer system (i.e. exchanging the hardware environment). If the hardware of both computer systems is similar enough, the software environment on the disk should perform identically on the new system with little or no adaptation.

By choosing a virtualization or emulation strategy the focus of preservation is on software environments (respectively disk images thereof) and their relation to and interaction with hardware, treating software environments as a runtime for software-based artworks. Hardware will inevitably suffer from technical and physical decay, and for cost reasons can only be preserved in individual cases (even then eventually failing). In contrast, preserved software environments change in neither their hardware requirements nor performance features over time. As a consequence of this, a particular preserved software environment can be considered as constant and stable in its requirements. The goal of an emulation strategy can be described as ensuring software environments (disk images) *run anywhere* and *run forever*. *Run anywhere* essentially means making a complex software installation portable. To achieve this, it is important to create the most generic software environment possible, particularly with regards to hardware dependencies. This should be such that a digital artefact is able to perform outside of its original environment. *Run forever* can only be achieved if the disk images and in particular their hardware dependencies are maintained over time. To achieve this, both the technical interfaces and external dependencies of disk images must be monitored and verified. If an interface's functionality breaks or an external dependency becomes unavailable, the disk image (and the software environment it contains) or the artefact itself must be adapted so they can perform again in the changed technical environment.

For the remainder of this section we assume that the artefact itself is either already available as a file or bitstream or the artefact is part of a disk image. Furthermore, we assume that the artefact's significant properties have been assessed and are available for verification purposes.

Acquiring Software Environments

This sections builds on the previous introduction to the characteristics of a digital object, and describes how to create disk images. As a first step to an emulation-based preservation strategy, the disk images to be connected to an emulator or virtualizer must be acquired. In the following sections two – non-mutually exclusive – acquisition strategies are discussed.

The task of determining an artefact's software environment, starts with the analysis of the artefact. The goal is to produce a stable and accessible setup which does not rely on the availability of physical hardware components, and to gather enough information to support future preservation tasks (i.e. ensuring long-term access to the artefact). The information needed will depend on the artefact's composition and on which individual technical components are present (outcome of *Question Set 2*), in particular:

1. a binary object (e.g. an executable application or its installation files),
2. its configuration (runtime configuration, such as application specific settings),
3. the object's source code,
4. any kind of documentation,
5. a reference installation (i.e. software correctly configured on a suitable computer system);

The existence (or not) of these components will affect the preservation options available. Having the artefact's source code available (enabling the recreation of the artefact on a technologically current machine) allows for a higher degree of technical abstraction and may also open the door to alternative strategies (e.g. a traditional migration approach). If source code is not available, creating an emulated version of the work may be the only suitable choice. How this emulated version is then created will depend on whether the software can be re-installed and whether all the required software dependencies are available and operational. For an emulation-based preservation strategy then, the artefact's software environment needs to be determined, in particular its composition and technical interfaces to the hardware layer.

To capture (or re-create) a runtime environment for a digital artefact, any emulation-based preservation strategy is likely to result at some point in managing a set of software environments as virtual disk images containing instances of these software environments. Disk images might either contain the digital artefact (or parts of it) or be prepared separately for a digital artefact held on separate virtual media (or similar). Two different and complementary workflows/strategies can be applied to the task of image acquisition, depending on practical options and requirements, and information available about the artefact and environment.

An emulation-based preservation strategy is based on the creation of disk images containing software environments. These can be created through two processes: either creating an image directly from the available source hardware (e.g. an artist's computer hard-drive); or creating a preservation disk image, where the software environment is manually constructed through the installation of necessary software components.

In the first process, the more information that is available about the software environment and its configuration, the lower the risk involved.

For the second process, the recommendation is to re-use and to adapt existing images to minimize variations on drivers and hardware used. This will simplify later migration of disk images between emulation platforms.

Recommendation: If possible, create two disk images, one imaged directly from a physical drive (if available) and another created by installing the software environment to reduce migration risks.

Preparation

Before imaging the media of a computer system, various options affecting the final image should be assessed. The levels of technical documentation already available depend on a series of factors, primarily whether or not the artist and/or the artist's programmer has supplied any technical details about the software, and whether they are available to answer any questions about the work's technical makeup, runtime process and system configuration.
In a first step, all components of an artefact are assessed regarding information about the artefact's technical dependencies, i.e. software, hardware and external dependencies and to support the selection of virtual hardware (cf. *Question Set 3*).
In a second step, a more detailed hardware analysis should determine if there is a risk that a specific hardware configuration may hinder booting the disk image of an artefact's machine. An example would be an incompatible storage interface (bus), such that the booting OS is unable to load or mount the root file system. To ease this work a controlled modification/adaptation of the original system's configuration (usually the OS's configuration and/or driver settings) can be advisable. For example, drivers and/or hardware extensions which are likely to prevent a direct boot should be removed or disabled. Typically, high-end graphics cards and incompatible storage controllers cause the most problems, as they are vital components in allowing access to the root file system and in enabling graphical output. If such modifications are necessary, it is advisable to clone the original hard-disk (in order to be able to track and document modifications made).
For the next step, imaging targets need to be assessed. Ideally, all three conceptual layers should be sufficiently abstracted so that each layer can be mapped to an individual artefact (virtual media) or conceptual entity (virtual hardware). For instance, the unique parts of a digital artefact (i.e. the digital artwork) should be isolated, stored and managed independently of their software environment (which itself contains non-unique, often commercially published, software (c.f. Fig. 5). Given this aim, the goal of the imaging workflow is to create two distinct disk images forming individual representations, one containing the digital artefact and one containing the required software environment to be used with the physical or emulated computer hardware. For this to be carried out, the digital artefact requires isolation/abstraction from its rendering environment, through the substitution of existing and installed software components with abstract software dependencies. For the purpose of rendering the digital artefact, a virtual computer system is constructed (an emulator with a disk image attached to be booted) and the artefact is *injected* again into the environment. This yields an abstracted artefact, which may not only be rendered by its original software environment, but also by any other environment providing the well defined software dependencies. Tools are available to support the process of determining software dependencies for a digital artwork, to a certain extent. Runtime analysis tools like PET[11] or PMF[12] are able to extract software dependencies while the artefact is being executed, while databases like NSRL[13] can be used to map files found on the disk to software packages. Both methods are discussed in more detail in the section on re-creation of disk images.

[11] The Pericles Extraction Tool PET, https://github.com/pericles-project/pet (online, last access Nov. 11, 2016)
[12] Process migration framework, http://ifs.tuwien.ac.at/dp/process/projects/pmf.html (online, last access Nov. 11, 2016)
[13] National Software Reference Library (NSRL), http://www.nsrl.nist.gov/ (online, last access Nov. 11, 2016)

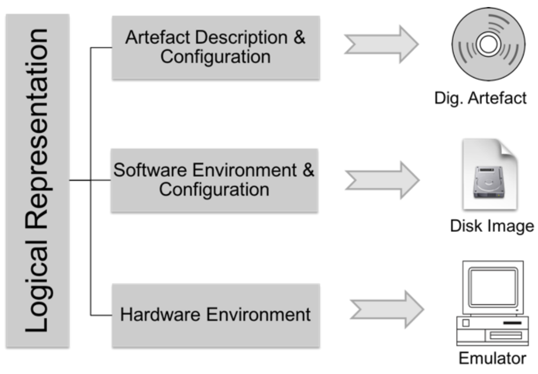
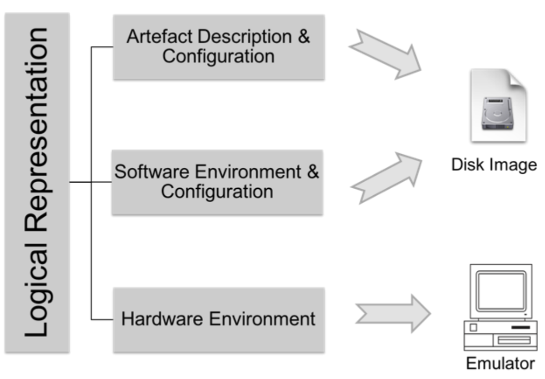


Figure 5: A three-layered representation of artefact and environment; here the digital artefact (i.e. artwork) could be isolated from generic software and hardware components.



Logical Representation
Artefact Description & Configuration
Software Environment & Configuration
Hardware Environment
Disk Image
Emulator

Figure 6: Another three-layered representation of artefact and environment; here the artefact (i.e. artwork) and generic software components could not be separated.

In some cases, the abstraction of the digital artefact is not possible, either for technical reasons, lack of case knowledge (e.g. if the creator of the environment is not available anymore) or lack of resources. In this situation the disk image is preserved as it was found, including the digital artefact. Due to the lack of an additional abstraction between the artefact and the software layer, the software environment cannot be replaced by an alternative rendering environment, reducing the options for risk management and preservation (c.f. Fig. 6).

In order to reduce complexity and potential failure of follow-up processes, it is important to prepare the computer system before imaging. For example, through removal or alteration of those configuration settings known to be unsupported by available virtual machines. Technical abstraction is the key to the longevity of a digital artefact. The greater the level of abstraction of technical dependencies, the more options for preservation actions are supported.

Recommendation: Create a disk image of the supplied system before introducing any changes. Simplify the hardware configuration and separate an installation into as many (conceptually meaningful) individual components as possible, so that you have disk images for each of them. The higher degree of technical abstraction that results will help support preservation strategies.

Virtual Media Imaging

Disk imaging describes the task of creating a raw, bit-identical copy of a physical medium (e.g. hard disk, floppy disk, CD-ROM). A bit-identical copy process reads any accessible byte from a medium and stores it in identical order in a flat image file. At this level there are no semantics or higher concepts associated with any bit/byte read. Successfully copied bytes may belong to a file contained on the medium or may be part of unused disk-space allocation. Hence, a *raw*, uncompressed disk image always has the size of the total capacity of the original medium imaged, regardless of the actual disk space used by meaningful data. For the capture of fixity information, there are two principle options:

- Fixity information can be created after the disk imaging process has finished, and then stored separately. For example as an individual file or a metadata entry as part of a repository.
- Fixity information can be created as an integral part of the imaging process and embedded within the image file. Disk image formats with such capabilities are usually called *forensic disk images* or *forensic containers*. A popular example with good open source tool support[14] and an open (partly reverse engineered) specification is the EnCase Expert Witness Format (EWF)[15]. A collection of tools (and training material) for both imaging options are provided by the BitCurator[16] project.

There are more efficient disk image representations (e.g. *sparse images*), which are able to grow in size with the actual disk-space usage. This feature, however, is only available for artificially created disk images, which are built from scratch using a virtual machine or emulator. In this case, an empty disk with a theoretical capacity is created and attached to an emulator or virtual machine, so that the image file grows as further disk space is required (e.g. installation of additional software).
Additionally, some disk image container formats package technical and descriptive metadata with disk images. For instance, the comprehensive Open Virtualization Format (OVF) standard[17], allows the packaging of configuration information, disk images and fixity information for a virtual machine into a vendor neutral format. Most virtualisation systems and some emulators are able to natively use OVF packaged virtual machines. A different type of disk image container offers the open VMDK format[18]. The VMDK format defines a header section containing technical metadata (e.g. if it is a sparse[19] or a flat/raw image). Disk formats like VMDK, VDI or QCOW may also contain different layers (versions or snapshots) of an image.
There are tools available (notably *qemu-img*) to convert between different container formats and to convert a RAW disk image to a container format as well as extracting a RAW image from a container. Furthermore, there is tool support to enable emulators to use arbitrary image formats or container formats, regardless of their natively supported formats. A version of *xmount* extended with QEMU's block driver code[20] is able to mount most available disk image and container formats and converts them on-the-fly to the expected format of the emulator. Even though image containers provide advantages (packaging metadata and image into a single representation), in that all relevant container formats are open standards and there is sufficient tool support, there are still some issues associated with using such complex image formats:

- To create these complex formats image conversion and management tools must be used, and this adds further levels of dependency which would have to be monitored.

[14] The libewf project, https://github.com/libyal/libewf/, (online, last access Nov. 11, 2016).
[15] EnCase image file format, http://www.forensicswiki.org/wiki/Encase_image_file_format
[16] BitCurator project, http://www.bitcurator.net/bitcurator/, (online last access Nov. 11, 2016).
[17] Open Virtualization Standard documents, http://www.dmtf.org/standards/ovf, (online last access Nov. 11, 2016).
[18] VMDK virtual disk format specification V1.1, http://www.vmware.com/app/vmdk/?src=vmdk (online, last access Nov. 11, 2016)
[19] A sparse disk image requires only disk space for allocated blocks, ie, logical disk blocks in use.
[20] Modified version of xmount – part of the EaaS framework, https://github.com/eaas-framework/xmount, (online last access Nov. 11, 2016).

- The benefits of containers are limited in the long-term, as preservation systems should in principle provide a reliable infrastructure to manage metadata and fixity information. This is especially true if each image and container format can be broken down to a RAW disk image, which is the common technical foundation format, representing the original content and structure of the physical media.

When imaging physical media, the use of a write blocker[21] is essential to prevent accidental modifications. If no write blocker device is available, the Bitcurator-tool-suite (a Linux-based live-system) offers an easy-to-use software-based safe-mount option[22] to achieve this.
There are many free and commercial tools supporting the imaging process, namely basic imaging tools for Unix-based systems, for which user-friendly front-ends are sometimes offered (e.g. predominantly based on *dd* – see also the section on tools). Guymager[23], also part of the Bitcurator suite, is an example of a user friendly application for imaging media. Step-by-step instructions are available in the Bitcurator Wiki[24]. A Windows-based commercial alternative is the Forensic Toolkit FTK.[25]

Disk images are a complementary concept to virtualization and emulation, as disk images represent media necessary to provide a data stream to emulated or virtualized computer systems. A disk image may represent: an OS or software stack which can be used to render a digital artefact; a copy of an artist's computer including both runtime software and one or many artefacts; or a single artefact e.g. an image of a published CD-ROM.

Recommendation: Simple, vendor-neutral disk image formats should be preferred over container formats, and metadata and disk images are better managed by an external preservation and storage infrastructure. This allows more flexibility on choosing a metadata representation and avoids duplication of effort.
If no forensically secured image is required then the recommended format is raw.

Selecting an Emulator

Emulators and virtualizers provide only a limited selection of supported hardware components. Their capabilities are best described by a list of supported *computer systems* (e.g. x86 ISA PC or Apple Macintosh Performa) and a list of supported operating systems. The most important information to be documented in the target system is therefore the general hardware architecture (e.g. CPU type, bus architecture or ROM type/version), in order to help choose an appropriate emulator or virtualiser. The choice of an emulator or virtualiser also requires information about the software environment (e.g. the emulator must support Windows 3.11), as even if an emulator supports a particular computer system, it may not support - or support only partially - an apparently compatible operating system (a popular

[21] http://www.forensicswiki.org/wiki/Write_Blockers (online, last access Nov. 11, 2016)
[22] Instruction on safely mounting media, http://wiki.bitcurator.net/index.php?title=Safely_Mounting_Devices,(online, last access Nov. 11, 2016)
[23] Guymager imaging software, http://guymager.sourceforge.net/ (online , last access Nov. 11, 2016)
[24] Step-by-step instruction imaging media using Guymager, http://wiki.bitcurator.net/index.php?title=Creating_a_Disk_Image_Using_Guymager (online, last access Nov. 11, 2016)
[25] Forensic Toolkit FTK, http://accessdata.com/solutions/digital-forensics/forensic-toolkit-ftk (last access Nov. 11, 2016)

example is a current version of Apple’s OS X, built to run on x86-based hardware but usually not supported by x86 emulators). While detailed hardware information will only rule out incompatible emulated computer platforms, the final decision on a suitable emulator/virtualizer requires that support for both the computer system and the associated software environment (primarily operating system support) are assessed.

A further, more detailed comparison between the identified hardware components and the features of a specific emulator is useful to estimate up-front the work required to migrate the machine’s software environment (disk image) to a new hardware environment. A detailed list of hardware components that are installed, provides insights on how the operating system was configured. For example, by comparing a detailed list of hardware components with a list of the hardware supported by an emulator it is possible to predict if a software environment (in form of the computer’s disk image) will boot directly using emulated software. If the system is able to boot to a certain point (e.g. a simple ‘safe’ environment with all non-essential hardware features disabled), built-in operating system tools can be used to adapt the system to the new, emulated hardware environment. These adaptations could involve identifying available hardware or suggesting drivers.

If it is not possible to boot the software environment from the disk image of a system, the disk image may require additional preparation and/or modification (where possible). Another approach is to rebuild the whole environment using an emulated computer system.

The importance of specific hardware components can only be assessed using information about the artefact and/or its software environment. For instance, if the hardware setup involves a high-end 3D graphics card, its importance to the whole installation can be judged on how the graphics card is used: is the performance of the card a key factor or does the software depend directly (i.e. direct hardware access) or indirectly (i.e. through an abstraction layer such as DirectX or OpenGL) on specific hardware features.

The necessity of incorporating software environment information (to assess the role of hardware components) as part of a digital artefact’s technical environment highlights the importance of the technical interfaces between software and hardware environment. These pose high risks to the artefacts functionality. These interfaces may break if an emulator drops support for specific hardware (e.g. emulator upgrade) or if an emulator becomes unavailable. In order to ensure long-term availability of a digital artwork, these interfaces need to be monitored and potentially replaced on the hardware (e.g. by using a different emulator or substituting emulated hardware) and/or the software side (e.g. exchange of OS driver or its reconfiguration).

Generalization

If the original computer system is available, images of physical media (e.g. a hard disk) can be made. To make these useable in an emulation environment and to facilitate the long-term management of disk images, as a first step it is necessary to *generalise* the technical interfaces between hardware and the lowest software layer (typically the OS, respectively OS hardware drivers and configuration). In case of disk images originating from physical media, generalisation is part of *migrating* a software environment from physical hardware to virtual/emulated hardware.

Generalising means that any specific drivers (for instance, from a disk image made from a physical computer) are replaced with drivers supported by the virtualization or emulation platforms in use. As part of this process, the choice of emulated hardware and drivers should

be consistent, so that for each combination of OS and hardware platform always the same (emulated) hardware component is used. Hardware dependencies should be systematically determined and all necessary adaptations should be kept to a minimum when migrating to virtual / emulated hardware, e.g. to maintain the environment's authenticity.
By generalising a software environment (respectively its instantiation as a disk image) the hardware dependencies and the number of drivers used among similar environments are reduced to a minimum. This means that for each emulated hardware platform and all associated software environments there is only a limited number of technical interfaces to be maintained and monitored, and this consequently means that the same migration strategy can be applied to different software environments. For example, ten different physical computers may use ten different video cards, which may each use different drivers with the same functions. By generalising the disk images created from these computers the number of drivers needed for a video card can be reduced, so that instead of ten different drivers only one is needed, and later on only one may need to be replaced (if necessary at all) for migration to another emulator.
As a result, the generalisation workflow produces a disk image to be used with a contemporary emulator. Additionally, implicit external and hardware dependencies are uncovered by running the artefact and its software setup in a different virtual environment. In particular, the image's technical interfaces can be documented by running on well understood virtual hardware.
To ensure that the significant properties are not affected the initial process of image generalisation should be performed during or after image acquisition, and preferably a comparison between the original and generalised systems should be made.

Structured Re-creation of Images

As an alternative to imaging physical media directly, images can be re-created for preservation purposes from scratch by, for example, manually installing a desired software environment in a virtualized or emulated environment.
As a precondition for re-creating environments, software components (e.g. operating system, drivers, applications etc.) which are required to render a digital artefact need to be available. Ideally the required components will already be packaged in a dedicated software archive, such that each software component can be added to an emulator setup and directly installed on the disk image. Ideally, the preserved software package will have been enriched with additional information such as references to operation manuals and installation procedures, license keys and technical information.

Automated software dependency determination

A second precondition to re-creating a software environment is knowledge of the necessary software components to be installed. One option is to make use of dedicated tools to determine an artefact’s technical dependencies, in particular software and libraries actively used while performing. This however, requires that the tools be installed on the original computer (which in some cases may be deemed too invasive) or the existence of an emulated version for analysis (e.g. using images of the computer’s media).
As an example, two comprehensive frameworks for dependency identification are presented:

- The *Process Migration Framework* (PMF)[26], was originally developed for repeatable and reusable scientific workflows and business processes and is an outcome of the EU FP-7 TIMBUS[27] project. The tool is Unix-based and, for the most part, uses *strace*[28] to identify software dependencies of a running process. The output of basic analytical tools is transferred to an OWL[29] context model to structure abstract dependency information and support algorithmic reasoning. The framework also supports the translation of abstract dependency information into installable software packages, though only for debian-based Linux distributions. [30]
- The Pericles Extraction Tool (PET)[31], an outcome of the EU FP-7 PERICLES[32] project is designed to "extract significant information from the environment where digital objects are created and modified." Also used during runtime, PET is a Java application which uses different *modules* to capture environment information of different types. A similar toolset to PMF, used for Linux-based systems, PET additionally supports further platforms, such as Microsoft Windows. For identification of installed software packages, the standard software management interfaces of modern operating systems are queried, with the application currently supporting Microsoft Windows, Mac OS X and Linux (RPM- and DEB-based distributions). Currently there is no modelling support for dependency abstraction, and no correlation of information on installed packages with software requirements for a specific artefact.

The outputs of runtime analysis tools can also be used to verify the results of a hardware migration process. For instance, by comparing the output from a newly instantiated environment and the output of the original hardware system. However, these results may be difficult to read as even minor variations may result in conflicts. The significance of any variation for the artefact's performance can be difficult to predict in an automated way, as tests typically only cover the equivalence of technical interfaces, not their general functionality or performance, and even then only in a virtualized or emulated environment.
An alternative, off-line approach to identifying installed software on a medium is to fingerprint files (i.e. calculate their hash or checksum values) and compare the results against software databases, such as the National Software Reference Library (NSRL)[33]. No direct correlation can be made between a digital artefact's software dependencies and the installed software identified, as there is no runtime information available. Nevertheless, this approach is helpful to distinguish between files belonging to the operating system or installed software (generic, non-unique files) and files probably created by the computer's users. This information can be

[26] Process migration framework, http://ifs.tuwien.ac.at/dp/process/projects/pmf.html (online, last access Nov. 11, 2016)
[27] Timeless Businesses and Services TIMBUS, http://timbusproject.net/ (online, last access Nov. 11, 2016)
[28] strace manual, http://man7.org/linux/man-pages/man1/strace.1.html, (online, last access Nov. 11, 2016)
[29] Web Ontology Language (OWL), https://www.w3.org/2001/sw/wiki/OWL, (online, last access Nov. 11, 2016)
[30] Andreas Rauber, Tomasz Miksa, Rudolf Mayer, Stefan Pröll, Repeatability and Re-usability in Scientific Processes: Process Context, Data Identification and Verification. DAMDID/RCDL 2015: 246-256
[31] The Pericles Extraction Tool PET, https://github.com/pericles-project/pet, (online, last access Nov. 11, 2016)
[32] Promoting and Enhancing Reuse of Information throughout the Content Lifecycle taking account of Evolving Semantics PERICLES, http://www.pericles-project.eu/, (online, last access Nov. 11, 2016)
[33] National Reference Software Library NRSL, http://www.nsrl.nist.gov/, (online last access Nov. 11, 2016)

used for isolating an artefact and separating (i.e. abstracting) it from its original software environment; for example, to render it in an alternative environment.

Management of software dependencies through image re-creation

To ensure long-term access to digital artefacts through emulation, not only availability of software and technical metadata is required, but environments also need to be tested and evaluated by conservators/digital curators who are aware of the artefact and its software environment (including properties and performance). A structured workflow is proposed here, specifying how to manually recreate a software environment, and allowing the preview and evaluation of the result of each installation step. Figure 7 shows a functional flow diagram of the workflow.

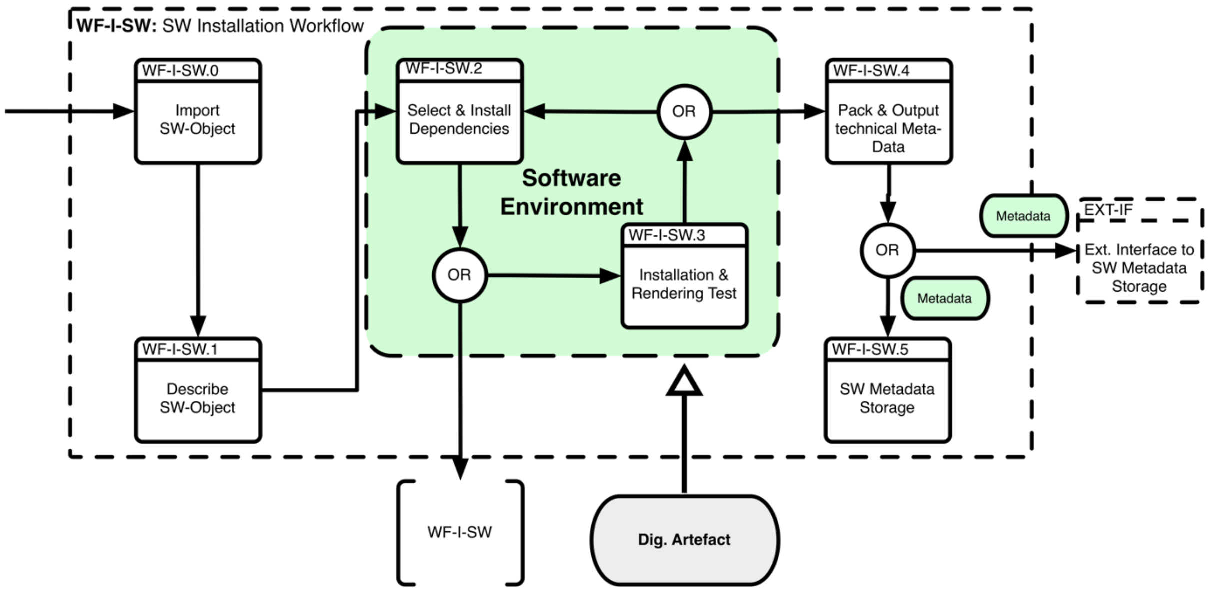


Figure 7: Structured workflow for re-creating a software environment[34]

The software installation workflow starts with importing a single software component; for example, a rendering application identified as on the artefact's software dependencies list (WF-I-SW.0). As part of this step the user is able to provide detailed descriptive and technical information about the software to be installed; e.g. describing the software's rendering capabilities in terms of supported file formats, version, supported languages and licensing (WF-I-SW.1).

In a second stage of the workflow, the software component's dependencies are determined, typically by following installation instruction of the primary software component to be installed. If a required dependency is not known or not yet available in the software archive, it must first be ingested into the software archive by using a recursive invocation of the installation workflow for this missing dependency software component.

Through an installation and test procedure (WF-I-SW.3) the software component's functionality and completeness (e.g. software components are installable, all dependencies

[34] Adapted from: Rechert Klaus, Valizada Isgandar, von Suchodoletz Dirk, Future-Proof Preservation of Complex Software Environments, Proceedings of the 9th International Conference on Preservation of Digital Objects (iPRES 2012): 179 - 183

are met and individual software components have no conflicting dependencies) is verified in this intermediate environment. For each successfully installed software dependency, the software environment's description is extended accordingly. Furthermore, the artefact to be rendered is made available within the newly created environment and its performance is tested (if possible). The resulting software environment then represents a suitable, manually tested and validated software environment to render a specific digital artefact.
While the proposed workflow requires significant manual user interaction and/or seems costly and time consuming, the workflow is able to create useful metadata describing software components and their own nested software dependencies. Following this process ensures not only that all dependencies are documented but also that all dependencies are archived; essentially forming an archival workflow for software. By focusing on generic software components and software environments, and their re-creation, the workflow's output (in particular the metadata, but if permitted by licensing also software and ready-made environments) may be shared between institutions and thus may be able to reduce preservation costs. Furthermore, metadata might also include user feedback about the performance and/or costs of a software component in a specific scenario.
Another useful side effect of this workflow is the creation of reusable, intermediate software environments and their instances in form of disk images. For any successful installation step, a new environment is created and documented. These intermediate environments can be stored efficiently as so called *snapshots* (as part of a VDMK, VDI, QCOW image) or *derivative disk images* using copy-on-write technology, such that only changes on the block layer are tracked. These images then, are not self-contained but instead refer to their *base images* for unmodified blocks. While a file system-based approach offers convenient tools to track changes of individual files, the metadata required to reconstruct these changes is rather complex and operating system specific. Using a simple, block-oriented approach of maintaining a virtual disk's changes has some advantages in a digital preservation scenario, due to its simple metadata structure. The result of changed blocks are just entries in a block mapping table (c.f. the listing below, showing an example using the QCOW image file format) which defines from which image the data should be read from. This simple representation allows a manual reconstruction, even if the original implementation is not available anymore.

```
Offset      Length    Mapped to  Image file
0           0x10000   0x270000   derived.qcow2
0x10000     0x10000   0x60000    base.qcow2
0x10000000  0x10000   0xab0000   base.qcow2
0x20000000  0x210000  0x50000    derived.qcow2
0x20210000  0x800000  0x2b0000   base.qcow2
0x30000000  0x10000   0xac0000   base.qcow2
0x3ffe0000  0x20000   0x80000    base.qcow2
```

The intermediate disk images may become useful for rendering other artefacts or creating similar environments. For instance, once an operating system and some basic utilities have been installed and configured, this environment can be used as a base image for different scenarios.
In order to automate such processes, information generated by unattended user interactions with a computer system can be used to generate automatic dependency installation

information. So called interactive session recorders are able to record user interactions such as mouse clicks/movements and keystrokes performed by the user during the interaction with an operating system and save them to an interactive workflow description (IWD)[35] file. These recordings can be played back if needed; for example, to automate software installations.
The task of re-creating disk images can be postponed for a certain time period if the artefact can be rendered in an alternative environment, or if there is a lack of tools or funding. Over time, however, there is an increasing risk of losing knowledge on old computer systems, making these tasks more difficult or impossible altogether. The structured production of base environments therefore, should be done in a timely manner and as close to the creation of a digital artefact as possible, since contextual information and usage knowledge of software is then most readily available. The emulation of software environments then, is able to provide a *view into the future* on access options, as well as highlighting potential rendering or performance issues.

Automated dependency identification and environment verification are more cost efficient than manual methods.
For the purpose of an emulation strategy however, these tools still have limitations and need to be combined with other methods. These tools work best in current and widely used environments, but are difficult to apply to older or very specific environments.
Manual environment re-creation provides metadata and a structured list of reusable components with manually verified properties. However it also requires significant initial effort, knowledge of old software environments, and a suitable software archive.
For maintaining disk images originating from a physical device a software preservation strategy (and associated software archive) is advisable. For (re-)creating environments it is essential.

Recommendation: Implement a comprehensive software archive (and software archiving strategy) focused on the software dependencies of digital artefacts. Make sure preserved software components are installable (useable) and complete (regarding their own software dependencies).

Maintenance and long-term Preservation of Disk Images

The outcome of a software environment acquisition workflow is a disk image (representing an instance of a software environment). Even though the technical file format of the resulting disk image is identical in all strategies (usually a virtual hard-disk, bootable by an emulator) different workflows must be followed for maintenance and long-term preservation. The actual available preservation actions will depend on information about the content and configuration of the disk images, from technical interfaces to hardware components, rather than on their file format. In order to maintain a software environment's *run forever* property, its technical

[35] cf. Latocha Johann, Rechert Klaus, Echizen Isao, Securing Access to Complex Digital Artifacts - Towards a Controlled Processing Environment for Digital Research Data, Research and Advanced Technology for Digital Libraries - International Conference on Theory and Practice of Digital Libraries, TPDL 2013, Springer: 315 - 320

interfaces have to be monitored and software environments have to be adapted to a new hardware environment if necessary.

Having a (detailed) software environment description allows to focus preservation planning and preservation action activities on monitoring and managing the technical links between software and hardware environments. While these links are stable in the short term, emulators are also subject to the software life-cycle, and will become technically obsolete at some point. Then, if new emulation software is required, a new (emulated) hardware environment needs to be determined which meets the technical requirements of the software runtime. If the technical interfaces between hardware and software environment are well documented - as a vital part of a software environment description, all affected software environments can be determined, and the search for new emulators can be guided effectively. If no perfect match is found, the required adaptations of the affected software environments can be predicted in an automated way using this same documentation.

For long-term accessibility the software environment's technical dependencies require monitoring, particularly:

- monitoring of software components used, including availability of licenses and external dependencies;
- monitoring of existing and emerging emulation and virtualization technologies for continued support of identified technical interfaces;
- monitoring of external dependencies.

Through monitoring technical dependencies, the identification of an imminent risk of obsolescence may indicate the need to migrate a disk image. This strategy then becomes very similar to the one applied to simpler digital objects such as spreadsheets or word processor files. In general, to implement a migration strategy, objects are monitored regarding their technical support (and therefore proximity to technical obsolescence) and migrated to a new, more sustainable or current format if required. Ideally all the significant properties of these objects are preserved in the process.

In the case of digital disk images, the significant properties to be addressed by a migration strategy usually relate to the identified technical interfaces. The functionality of technical interfaces can be documented, monitored and tested automatically, at least to certain extent. For instance, the software-side (or driver) of a technical interface to a sound card can be verified through a generic test (i.e. that sound is produced for different formats and configurations). If the test is successful, then it is highly likely that any digital artefact using the same interface is also able to produce sound output. However, a general assessment of the emulator's functionality, in particular the equivalence of original and emulated CPU, is a much harder task[36]. Furthermore, computational performance features such as the rendered frame rate of graphics after processing, disk I/O performance, synchronous video and audio or even interactivity (for example, the latency between a user input event such as a mouse click and the system's visible reaction), would all need to be verified. Ensuring verification of these features makes it particularly important to begin preparation (including creation of

[36] Nadav Amit, Dan Tsafrir, Assaf Schuster, Ahmad Ayoub, and Eran Shlomo. 2015. Virtual CPU validation. In *Proceedings of the 25th Symposium on Operating Systems Principles* (SOSP '15). ACM, New York, NY, USA, 311-327. http://dx.doi.org/10.1145/2815400.2815420

appropriate documentation) and implementation of emulation-based preservation strategies while the original system is still available to compare against.

Preservation Risks

There are several levels of documentation possible for all three monitoring activities, depending on the choices made acquiring the software environment.

The highest level of risk for emulation exists when there is only limited knowledge about the software environment, its composition and configuration as well as hardware dependencies, i.e. technical interfaces. If, due to limited information about technical dependencies, an a-priori risk assessment is not possible, a future migration strategy may fall back to trial-and-error. Furthermore, there is a risk of an incomplete monitoring process, potentially missing obsolescence indicators. Similarly, there is a high risk of failing to rebuild the software environment if the software environment's composition and configuration is unknown. This risk increases significantly over time, as the knowledge of the computer system and the availability of the necessary software packages decreases. If there are resources available for technical analysis, documentation and collection software packages at acquisition, then long-term preservation risks can be lowered more efficiently. This is particularly relevant for hardware dependencies and the configuration of the operating system, for instance a complete list of all the drivers installed and description of the hardware configuration used. With this information at hand an a priori assessment of technical migration risks becomes possible, as necessary drivers can be collected in advance and potential alternatives considered.

The lowest risk level is achieved when complete documentation is available about the enclosed software environment and its hardware dependencies, such that the environment can be rebuilt from scratch if necessary. In this case a preservation strategy does not solely depend on the acquired disk image (i.e. a single, "fixed" setup), as multiple strategies can be applied simultaneously. The same applies to disk images which are specifically built for preservation purposes. These images were already built within a virtualized / emulated environment, so the risk at migration is reduced, as information on the images' content and configuration is - assuming a (semi-)automated documentation process - readily available and the installation procedures easily reproducible. The effort required for maintenance, however, may differ. This is due to different creation processes and creation goals.

The images created using the documented system requirements and installation instructions replicate an existing system as closely as possible. Depending on the granularity of the available documentation, there might be slight variations and alteration of the original system specification, for example, to cope with external dependencies and/or a different hardware options in the virtual machine. A further migration of these images to new platform may require an individualized strategy, as their similarity to the original system should be maintained.

In contrast images with software installations reduced to an artwork's essential software components are more resilient to technological change, as wider variations and adaptations are acceptable, as long as the artefact can be rendered, i.e. its significant properties remain intact. In both cases, reproduced images are able to share a common technological base, e.g. share an operating system installation and configuration as well as relying on the same technical interfaces. Through a shared technological base, preservation risks can be shared among similar artefacts and collecting institutions.

Managing external Dependencies

The management of external dependencies requires a set of strategies, as external dependencies are technically and conceptually more diverse than hardware found in computer systems and there is usually no drop-in replacement available. Due to the diversity of external dependencies, only abstract workflows are presented.

The first, and usually most efficient strategy is *internalisation* of external dependencies, such that they become either a part of the digital artefact or its software environment. In general, there are two types of dependencies which can be internalized, abstract data dependencies and functional dependencies.

A simple example for a data dependency is an artefact accessing data which is externally stored. An internalisation option is to copy/mirror the data and make it locally accessible, such that it becomes a part of the artefact. In general, this option is applicable for abstract data dependencies, with data being accessed or served through standard protocols, e.g. protocols directly supported by the OS. Most of the time, modifications to the object and sometime even to the software environment can be avoided. In some cases, however, changes have to be made, e.g. to point to the new location of the data (which is in particular problematic if the digital artefact used hard-coded URIs and the artefact cannot be changed).

For pure functional external dependencies, e.g. a computation is made by an external machine and the artefact requires the result to perform or a software dependency requires external functionality, such as a license server, or mixed functional and data dependencies (e.g. data is served through a specific protocol, which requires additional software support such as databases), can be internalized, if the (server) machine is available and suitable for emulation. The internalized machine can then be treated as a dependent, secondary artefact, emulated and connect to the primary artefact.

A second strategy to deal with external dependencies is *abstraction*, which should be included in preservation planning activities, as it may not always yield into direct useable results, but prepares the ground for future preservation actions. Through abstraction the risks of failing or obsolete components to the whole setup can be reduced. The main goal is to abstract technical requirements, such that equivalent replacements can be identified and applied. For instance, a problematic software dependency with a dependency on a license server may be exchanged with a less problematic one, e.g. a software product providing the same technical features for a given digital file format but does not rely on an external functionality.

Finally, *emulation & simulation* can be pursued, if other strategies fail or are not applicable. This strategy requires the broadening of the scope of emulation to include interfaces and behaviour of external components. For instance, if an artefact relies on the availability of a video-portal accessed through a dedicated web-service protocol, the protocol interface may be simulated and either translated to a newer protocol version to retrieve content or the whole service is simulated using e.g. recorded or archived content. An example of an emulated web service in the domain of research data management is provided by Miska et al. [37] A similar strategy can be applied to external hardware and hardware protocols.

[37] Tomasz Miksa, Rudolf Mayer, Andreas Rauber: Ensuring sustainability of web services dependent processes. IJCSE 10(1/2): 70-81 (2015)

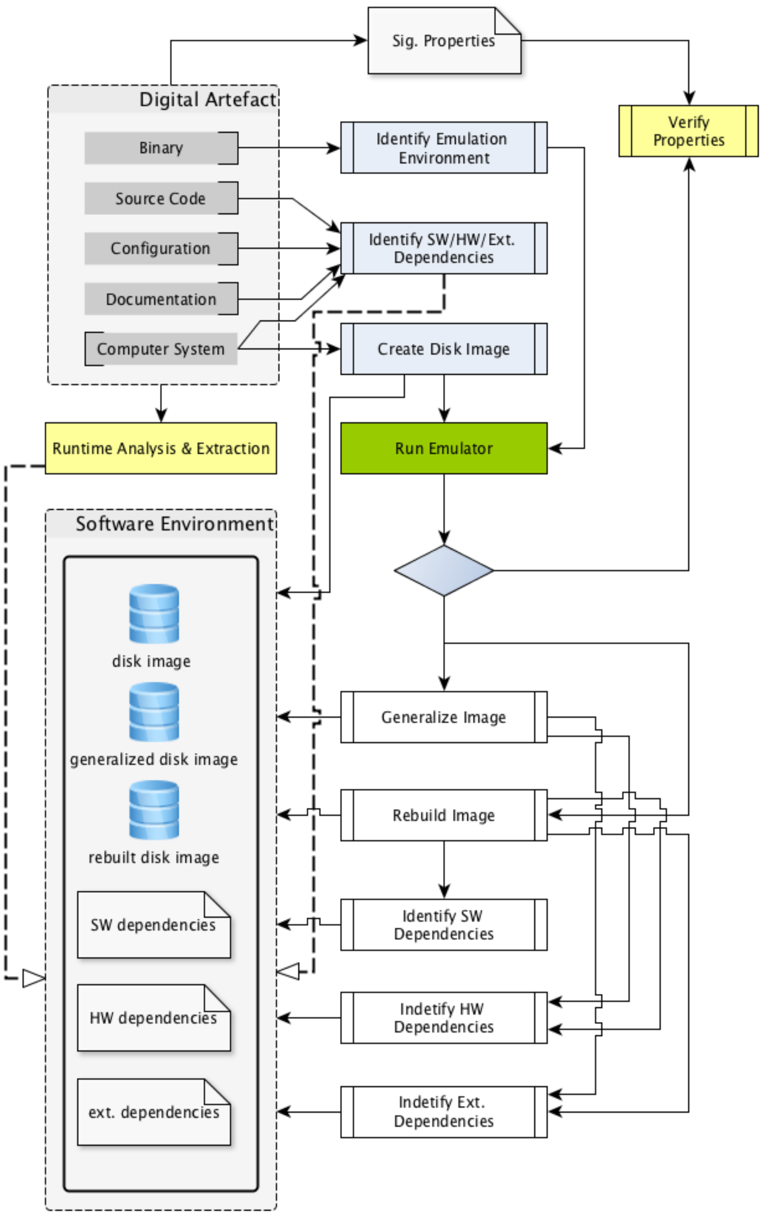
Maintaining virtual disk images is similar to a “classic” digital object migration strategy. The properties to watch for technical obsolescence are the technical interfaces between software used and (external) hardware components. Ideally, the number and variety of technical interfaces used by preserved software environments / disk images can be reduced to a minimum, to minimize the number migration steps required.

Recommendation: Develop a policy for substituting (migrating) technical interfaces, in particular, define which (technical) properties have to be maintained (e.g. preserve a general technical functionality - the ability to print - versus the availability of a specific printer type). Substituted hardware should be assessed to ensure its feature set and performance meets the artefact’s requirements.

Draft Workflow Proposal

This proposal is an initial draft of a possible workflow for applying an emulation-based strategy to the long-term preservation of software-based artworks. This is a schematic proposal, meant to guide the way the artworks are analysed, rather than provide a rigid set of instructions on how to preserve them.
There are two common situations when a new software-based artwork is acquired. In the first case, both the artist/programmer are available to answer any questions, and an installation in working condition is available as a reference. In the second case all that is available is a computer (or similar) with the operating system and the digital artefact installed. When migrating emulated environments, options will depend on the information available and whether you have a running version of the work as a reference.

Image Acquisition



Step 0: Determine if emulation is a preservation strategy to be considered for a given digital artefact (cf. Question Set 1)

- Determine if, in principle, the artefact can be used with virtual hardware
- Evaluate all available options regarding costs and feasibility
- Consider whether emulation is in accordance with the artist's view on the preservation of their artwork and whether emulation will maintain the work's significant properties[38] (please note this relates to more than the digital artefact itself).

Step 1: Identify available components and gather information about production process (cf. Question Set 2)

- Identify the binary (executable) parts of the artefact. Is there an installation procedure (either automated or manual and guided by documentation)
- Is a (still working) reference installation (computer system) available? If the artefact is not available as separate file(s), extract the artefact if possible
- Identify whether source code is available and if so, request it
- If possible, have the function of the digital artefact explained and documented in plain human-readable language (for simple software only)
- Determine the artefact's significant properties, if not already available. Based on these prepare for verification / assessment of the emulated version

Step 2a: Imaging of computer system (if available)

- Create a (back-up) disk image of the reference installation:
  - to use with emulators
  - to be able to run a runtime dependency extraction
- Document the reference installation (to be able to abstract technical dependencies on the software environment layer) in regards to:
  - Hardware and software environment
  - System configuration
  - Identify hardware and software dependencies
  - Document artefact performance

Step 2b: Identify an *Emulation Environment* for binary (executable) objects

- What technical platform is required (e.g. general CPU / bus architecture)
- Determine the minimal software environment (OS and additional software requirements)
- Identify and test emulators supporting both the required hardware and software dependencies

Step 2c: Identify dependencies based on available documentation, configuration and source code

- Determine the programming language, required tools and settings to (re-)compile the artwork
- Determine software, hardware and ext. dependencies based on the documentation, runtime configuration and source code
- Use runtime analysis tools (if available)

[38] As defined in Laurenson, Pip. 2015. "Old Media, New Media? Significant Difference and the Conservation of Software-Based Art", available at: https://www.academia.edu/24233428/Old_Media_New_Media_Significant_Difference_and_the_Conservation_of_Software-Based_Art (online, last access Nov 11, 2016)

Step 3a: If you have a contained digital artefact, re-build a disk image
Requirements: software archive and repository of base images

- Prepare for rebuilding image workflow (*Question Set 3a*)
- If artefact exists as a contained component, install on a new system, with no other software installed beyond the necessary
- Start with an existing OS (or other prepared "base" system available)
- Install missing software components, add new software to the software archive if necessary
- Document (additional) software dependencies and their configuration
- Identify additional (indirect) external dependencies and/or hardware dependencies, posed by newly install software components
- Determine and document the software environments final hardware dependencies using the emulators hardware description and configuration

Step 3b: If you cannot extract the digital artefact from the operating system create a generalized disk image

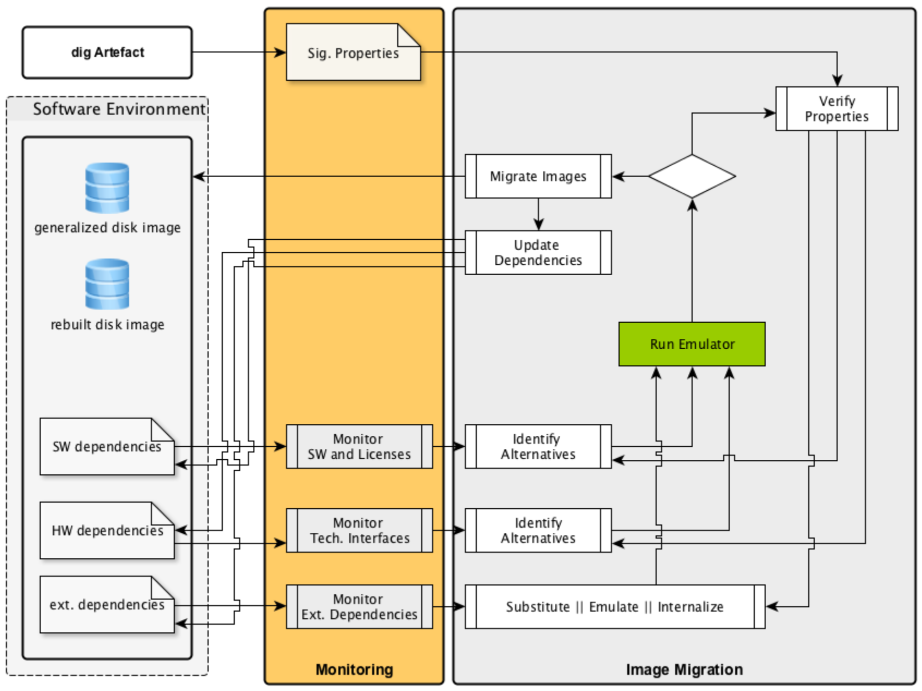
- Check preconditions for generalization process (*Question Set 3b*)
- Simplify/unify/generalise technical dependencies between hardware and software environment on a disk image
- Identify any drivers (and associated configurations) used as well as the hardware components used (to be able to document hardware features used by the software environment)
- Update / adapt driver configuration, add driver software to the software archive if necessary
- Identify implicit (undocumented) external dependencies
- Determine and document the software environments final hardware dependencies using the emulators hardware description and configuration

Step 4: Compare the newly created emulated version of the work to your reference version

- Confirm that it meets both the technical requirements of the artefact and its software environment and the performance features documented in Step 1 and 2.
- At this stage, if possible, the artist should be able to approve (or not) the emulated version. This step will help confirm that the significant properties (or work-defining properties) are maintained in accordance with the artist's intention, or that some of those characteristics are not maintained and therefore an alternate strategy may have to be found.

As a result of the processes above the software dependencies of the digital artefacts will have been identified. This information should be used to create a software archive of the operating systems, drivers, applications and specific libraries you may require.

Maintenance



Step 1: Monitoring (continuous task[39])

- **Significant Properties:** Significant and work defining properties should be observed and checked against those originally defined. This might be through a general assessment of the availability and usability of the technical environment necessary for a software-based artwork to perform. For instance, crucial technology may become unavailable, such as specific display types or special human-machine interaction devices.
  Monitoring significant and work defining properties is partly related to an emulation-based preservation strategy (e.g. the unavailability of non-computational technology may result in a failed emulation strategy), but in general out of the scope of a technology-focused monitoring activity. However, any preservation action triggered by emulation induced preservation action, requires a verification of the result through pre-defined significant properties.
- **Software Dependencies:** The content of the software archive has to be monitored for availability and usability. Internet-based license servers may disappear (see also monitoring of ext. dependencies) or changes to licensing terms may prevent use of some software components, as well as the availability of their technical interfaces, e.g. the ability to run software (see also monitoring of technical interfaces). Monitoring software dependencies can be implemented as a passive task, reacting to

[39] This is an emerging area of conservation practice. However, depending on the size of your collection, the type of technologies and rates of technological change it is considered prudent to carry out a general assessment of current risks as part of a regular meeting with relevant expertise (internal or external), for example annually.

licensing changes (e.g. cancellation of subscriptions, etc.). Further monitoring can be subsumed by monitoring technical interfaces and external dependencies.

- **Technical Interfaces:** The sum of all technical interfaces, either as a result of the collection of software installations (including OS, drivers, libraries and software) as re-built images or the inferred technical interfaces from generalized disk images, need to be monitored for availability and usability.
  Technical interfaces are endangered or break with the imminent obsolescence of the current software or hardware platform or their upgrade. Hence, the monitoring task should focus on software and hardware updates of the technical infrastructure providing emulation.
- **External Dependencies:** Monitoring external dependencies can be less clearly defined due to the diverse nature of potential external dependencies. Hence, it is difficult to define a monitoring schedule for these dependencies.

Step 2: Image Migration

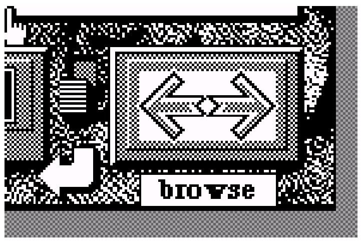
- **Identification of alternatives:** The unavailability of technical interfaces or software components usually requires that alternative software packages or alternative hardware emulators be identified. This also requires updated drivers and/or system migration.
- **Verification:** If alternatives have been identified, the individual digital artefact's properties have to be verified. If upon verification it is found that the artefacts properties were not maintained, then search for alternatives has to be repeated.
- **Migration of images / Update dependencies**: If an artefact's significant properties could be verified with the updated hardware or software environment, a new migrated image is created and documented, ie. updated dependencies.

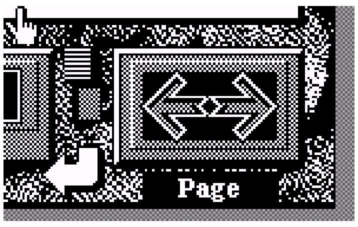
Limitations

In this report we have presented an approach which, instead of looking at work-specific properties, focuses on the digital artefact's technical dependencies. If emulators were able to perfectly reproduce out-dated hardware components, this technical perspective would be sufficient - at least concerning any computational aspects of the artwork. In practice however, emulators are far from perfect.

An emulation strategy for a specific work may fail due to a number of different issues. The most likely cause however, is due to incomplete or limited implementation of technical features e.g. emulated hardware supports only a limited set of functionality, or only supports some settings or configurations. Usually such limitations are revealed easily, through a simple test-run of the artefact in a virtual environment. In some cases, no adequate emulator exists.

A second class of technical limitations are far more difficult to uncover: bugs (implementation errors) affecting the program logic. These kinds of bugs may cause subtle changes of application's behaviour.





Example[40]: Beyond CyberPunk! - 1990 (Mac 7.6), running in different versions of Sheepshaver. "*[...] a multi-screen/page text, and you have a button in the lower right corner to change the navigation mode to 'page' mode instead of 'browse' mode. Click on this button to set 'Page' mode, then click on the right arrow to go to next page. The next page is displayed, but the mode has changed back to 'browse'! In the original version [*rendered on the original hardware*], the mode stays to 'Page'.*
The same (faulty) behaviour can be observed using OS 7.5 running on any version of BasiliskII.

Due to theses technical shortcomings, an emulation strategy relies heavily on the capability to verify the performance of a digital artwork and the availability of its significant properties in a new technical environment. While some technical properties may be assessed in an automated way, for digital artworks automated testing has its limitations. Software-based artworks have a second (mostly conceptual) layer of significant properties, which cannot be tested in an automated way and require a specialist's assessment (for example, qualities of the artefact's behaviour). Hence, a structured verification of the accuracy of an emulations performance remains one of the most important challenges when implementing an emulation-based preservation strategy.

[40] Example provided by Yves Bernard, IMAL, Brussels, BE

Conclusion

The first step for the preservation of a software-based artwork is a technical analysis of the hardware and software setup required for its display. This analysis provides the basis for a description, which can be broken down into three conceptual layers: artefact and configuration; software environment and configuration; and hardware environment. Assessing preservation options for each of these layers individually, provides a differentiated view on technological risk and potential mitigation options, and helps to make a changing technological environment more manageable.

The higher the degree to which an artefact can be abstracted from its technical environment, the more options for preservation actions remain. If an artefact can be re-built for different technical environments, its software, hardware and external dependencies may be substituted (e.g. by changing the code or the artefacts setup).

An artefact's software environment is of particular interest as it usually connects digital objects with hardware. If the software environment is known in its composition and configuration, the environment can, if necessary, be rebuilt in order to mitigate risk (e.g. substituting problematic dependencies).

Furthermore, it becomes possible to consolidate disk images by, for example, building on a common base system consisting of operating system and a unified hardware configuration. Breaking down the technical characterization to a (common) set of technical interfaces shared among many artefacts of a similar type makes it possible to focus monitoring and technical migration work. If technical interfaces break, disk images may be migrated to a new (virtual) technical environment. Ideally, a migration path is only developed once and applied to all suitable artefacts.

The technical approach presented requires a wider supporting framework. Primarily, a dedicated software archive is necessary (which includes management of licenses) to help ensure that a given software environment can be rebuilt. Additionally, it is useful to maintain a testbed of typical environments and common technical interfaces to be tested on newly released emulators. In contrast to testing an artworks work-specific significant properties, these activities, and in particular the technical infrastructure, can be shared and re-used not only for sets of similar artworks but also among different institutions.

Terms and Definitions

the artwork	Refers to the combination of software, hardware and installation information required to reproduce the artist's intent. A comprehensive representation or description, including software, hardware and installation information or similar documentation.
digital artefact	The digital part of an artwork, excluding (non-custom) software (cf. software definition below).
software	Of-the-shelf (non-customized) software components, ie., software components likely to be found in a software-archive.
hardware	Computing hardware the digital artefact and its software dependencies is running on. This includes the "machine" (computer) and any hardware components "built-in".
peripherals	Hardware components attached to the machine, using external technical interfaces such as USB, serial port, FireWire, etc. Example peripherals: camera, projector, screens or generic input devices.
(rendering) environment	An instance of (emulated) hardware and software (installed and configured), suitable to render a specific digital artefact.
software runtime / software (rendering) environment	The software runtime conceptually captures all software components (applications, libraries and operating system), installed and configured. The runtime is then able to render a set of specific digital file formats, including executables.
emulation	Emulation imitates hardware and its functionality in software, e.g. a software program is behaving identical as an old computer system.
simulation	There is no strict distinction between emulation and simulation. Usually, emulation refers to imitating hardware, while simulation is often used when referring to imitating software or software interfaces.
sparse image	Sparse disk images allocate storage space (on the hosting disk) on-demand. A sparse virtual disk can be created with a (virtual) size of 20Gb but uses only a fraction of its size for storage initially. The storage requirement grows with actual demand.
raw image	A RAW image is a flat representation of a virtual disk image. A RAW image is of exactly the same size (file size) as its virtual capacity. Hence, no additional data has been added (such as header or fixity information) or is unallocated (as in sparse images).

Tools

Disk image tools

dd **ddrescue** **dd_rescue** **sdd**	Basic (unix-based) disk imaging tool to create images of block devices. Example (imaging from first physical disk (SATA): `dd if=/dev/sda of=/path/to/img.raw` *ddrescue* additionally is „trying hard to rescue data in case of read errors." *sdd* a dd part of the *Sleuth Kit (TSK)* http://www.sleuthkit.org/ Output formats: RAW
ewfacquire	Part of libewf/ewfutils (unix-based). http://code.google.com/p/libewf/ Disk imaging and image conversion tool (is capable reading from physical media and virtual media). Creates forensically packaged disk images (EnCase format). Output formats: EWF (EWF-E01, EWF-S01)
guymager	User-friendly frontend for various imaging tools (using-based). http://guymager.sourceforge.net/ Output formats: RAW, EWF/EnCase, AFF
FTK	Windows imaging tool. http://accessdata.com/solutions/digital-forensics/forensic-toolkit-ftk Output formats: RAW, EWF
BitCurator Live System	Live system, boots from CD/DVD or USB, also available as VM. http://www.bitcurator.net/ Contains and integrates most unix-based imaging and image analysis tools. Step-by-step instructions for many imaging tasks: http://wiki.bitcurator.net/index.php?title=Main_Page

Image management tools

qemu-img	Standalone utility, part of the QEMU code base. Is able to identify, create and convert between disk image formats / container formats, such as VDI, VMDK, VHD(X), VPC, (Q)COW2, RAW.
	
	Example (create a VDI from RAW):
	`qemu-img convert -O VDI test.raw out.vdi`

mount, fdisk	Unix tools, capable of analysing and mounting RAW images:
	
	Example (mounting the first partition):
	`mount -o loop,offset=32256 /path/to/img.raw /mountpoint`
	
	To find out the right offset (start of the partition):
	`fdisk -l /path/to/img`
	
	Adapt offset for second partitions etc.

kpartx	Unix tool to mount / manage disk images with multiple partitions.
	
	Example (list partitions, similar to fdisk -l):
	`kpartx -l img.raw`
	
	Example (prepare all partitions):
	`kpartx -a -v img.raw`
	
	Partitions are then available under `/dev/mapper/loopXpY` with *X* being the loop device (usually automatically chosen by the system) and *Y* the logical partition number.
	
	Having all partitions mapped as loop devices, individual partitions can be mounted:
	`mount /dev/mapper/loop0p1 /mountpoint`

qemu-nbd	Standalone utility, part of QEMU code base. Useful to mount / work with non-raw images. Supports image formats as listed for qemu-img.
	
	Example (mounting a VDI image):
	`modprobe nbd`
	`qemu-nbd -c /dev/nbd0 /path/to/image.vdi`
	`mount /dev/nbd0p1 /mountpoint`
	
	Remarks:
	- modprobing the nbd module might be necessary to enable kernel support. After this command, nbd device nodes become available (/dev/nbdX)
	- *qemu-nbd -c* “connects” the image (and its partitions) to a nbd device node. Choose a free node. To detach an image use *qemu-nbd -d <nbd dev>*

xmount	xmount is a FUSE driver that emulates different disk image formats (VDI/VHD, VMDK and, of course, RAW) without the need for a complete copy/conversion (e.g. using qemu-img). Hence, one can FUSE-mount a RAW image file as VDI using xmount.

xmount + qemu_block_drv	Enhanced xmount supporting more image formats (using the qemu block driver code) https://github.com/eaas-framework/xmount
	Example (mounting a QCOW2 image as a VDI):
	`xmount --in qemu writeback.cow --out vdi --cache writethrough --inopts qemuwritable=true /mountpoint`

Hardware analysis[41]

hwinfo	Windows freeware tool for hardware analysis. Many export options. x86 only. http://www.hwinfo.com/
HDT	Simplistic but comprehensive tool, suitable to be installed on floppys, CDs or USB boot media. x86 only. http://www.hdt-project.org/
Ultimate boot CD	Live CD with comprehensive hardware analysis, test and detection tools. x86 only. http://www.ultimatebootcd.com/

Image analysis tools

The Sleuth Kit (TSK) **Autopsy**	TSK is a framework (tool collection) for (forensic) analysis of disk images. E.g. using the *NSRL* database for file identification (e.g. standard non-user generated files). Comes with *Autopsy* - a GUI frontend. http://www.sleuthkit.org/
File Information Tool Set (FITS)	Java-based framework / wrapper for “classic” file identification and meta-data extraction tools, such as DROID, Apache Tika, Jhove, etc. http://projects.iq.harvard.edu/fits/home
BitCurator Live System	Packaged various data triage tools with good documentation: http://wiki.bitcurator.net/index.php?title=Main_Page#datatriage

(Selected / Recommended) Emulators / Virtualizers:

System	**Recommended Emulator(s)/Virtualizers**
DOS (and variants)	● DOSBox (for most DOS programs, in particular games)[42] ● Qemu (ISA-PC) ● VirtualBox
Windows 1.x - 3.x	● DOSBox ● Qemu (ISA-PC) ● VirtualBox

[41] Remarks: Hardware detection and analysis utils are platform specific - most available utilities are only available for x86-based computer systems. For non-x86 systems, in particular obsolete platforms, no specific recommendation is possible.

[42] More details on DOSBox’s hardware configuration can be found here: http://devtidbits.com/2008/03/16/dosbox-graphic-and-machine-emulation-cga-vga-tandy-pcjr-hercules/

Windows 95 and newer	● Qemu (KVM if supported) ● VirtualBox
Apple System 6 and older	● PCE (Apple 2) ● Kegs (Apple 2 GS) ● MESS (Many older Apple Systems)
Apple System 7	● Basilisk2
Apple System 8/9	● Sheepshaver
Apple OS X (PPC)	● Qemu (Power Mac G3)
Unix/Linux Variants	● QEMU (also for non x86 system - Sun SPARC, ARM, PPC) ● VirtualBox (x86 only)

Runtime analysis and packaging tools:

ptrace (as well as strace and ltrace) lsof	*ptrace*, *ltrace* and *strace* are low-level process tracing tools (unix-based, including Mac OS X - Windows pendants are available), able to monitor the behaviour of a process (programme) while running. The program can be interrupted to inspect its current state, library and system calls can be traced to determine system and software dependencies. *lsof* allows to determine which file a process is using.
PET	A runtime analysis framework, wrapping different analysis tools (such as ptrace etc.) to cover a broad range of use-cases and cross-platform support. *PET* is able to create a hardware information report, to determine software dependencies and to track file-system activity of a process. https://github.com/pericles-project/pet
Umbrella	*Umbrella* is a framework, able to describe and to recreate a process' execution environment and its data dependencies using container or virtualization technology. The process' execution environment is described through a hardware specification, operating system and software environment. To determine a process' runtime dependencies also *ptrace* is used. http://ccl.cse.nd.edu/software/umbrella/
ReproZip	Tool from the research data management domain, aiming to capture and package a scientific process, such that it can be independently reproduced. To do this, a running process is traced to determine runtime information and to package data, executables, libraries etc. in a shareable container. Currently only Linux commandline applications are supported. https://vida-nyu.github.io/reprozip/
PMF	The process migration framework (PMF) provides a workflow to extract a process from its original environment and to redeploy it in a virtual machine. Similar to other Linux-based runtime analysis tools, (*s)trace* is used to capture runtime dependencies. http://ifs.tuwien.ac.at/dp/process/projects/pmf.html

CRIU / Container (such as Docker, LXC, Rocker…)	*CRIU* is capable of freezing and serializing a running process (ie create a snapshot) and to restart the process later. In combination with the process container concept the “running” process can be preserved and re-started on a different machine (under certain conditions). Even though this approach has technical limitations, it offers similar technical metadata (description of technical dependencies and software dependencies). https://criu.org/Main_Page https://www.opencontainers.org/

Monitoring external interfaces

jpnevulator	*jpnevulator* is an open source tool to capture data on serial connections. Capturing serial data may be useful if there is external hardware connected to serial ports. The capture file can be kept as a reference for the machine - external hardware conversation. https://jpnevulator.snarl.nl/
wireshark	*wireshark* is “the” tool to capture network and USB data. Cross platform support. https://www.wireshark.org/ https://wiki.wireshark.org/CaptureSetup/USB
wsMonitor	*wsMonitor* is a Java-based monitoring tool, to monitor and capture SOAP messages from and to external Web Services. https://www.openhub.net/p/wsmonitor