Chapter 22

Editors' Notes: For this chapter on caring for software- and computer-based art, the editors of this book, Deena Engel (Clinical Professor Emerita, Department of Computer Science, Courant Institute, New York University, USA) and Joanna Phillips (Director, Düsseldorf Conservation Center, Germany) take on the role as authors and are joined by Tom Ensom (Tate/Independent, UK) and Patricia Falcao (Tate/Goldsmiths, University of London, UK). Deena Engel and Joanna Phillips draw from their five-year museum-academic collaboration to study and treat software-based works at the Solomon R. Guggenheim Museum, New York, where Phillips previously served as Senior Conservator of Time-based Media. At Tate, conservators Tom Ensom and Patricia Falcao collaborate towards improving the care of software based-artworks in the collection, drawing from practices outside the conservation field and synthesizing them to the institutional context.

With their combined expertise and experience, the authors step through an ideal acquisition process that enables the preservation of software- and computer-based artworks for the future. An in-depth look at the technical make-up of software-based art aims to serve as a foundation for its examination, documentation and treatment, and an overview of different conservation practices and treatment approaches is offered to inform the decision-making process towards sustaining the collection life of these artworks.

Caring for Software- and Computer-based Art

Deena Engel, Tom Ensom, Patricia Falcao, and Joanna Phillips

22.1 Introduction

Within the field of time-based media conservation, the newest area of specialization is the care for software- and computer-based art. While contemporary artists have been employing software and computers since the mid twentieth-century, the number of collected artworks in this category is still relatively small even in major international collections with large time-based media holdings. Over the last decade, however, as artists continue to embrace the latest digital technologies in their work and collectors are growing more confident in their ability to care for these works, an ever-increasing number of software- and computer-based artworks are entering art collections. The wide variety of these works ranges from web artworks viewed in a browser to generative animations on the screens of Times Square, from robotic arms performing in the gallery to game-like apps on your mobile phone.

However diverse in experience, these works have many commonalities with other types of time-based media in regard to their conservation: they must be understood and treated as dynamic systems of interdependent hardware and software components; their industrially produced technologies may be replaceable and/or variable; and their inherent change has to be managed carefully to prevent damage to their integrity. But the extreme complexity of their systems and their accelerated obsolescence and failure can make their care more challenging than caring for "traditional time-based media works" (Laurenson 2014), and the specialty expertise needed to maintain and treat them can be very specific.

To address these challenges and support the collection and longevity of software-based art, different communities, initiatives, and institutions have been researching and developing new conservation practices and conceptual frameworks in recent years. Contemporary art museums have launched dedicated projects such as the *Conserving Computer-based Art* initiative at the Guggenheim Museum in New York (Guggenheim Blog 2016, Guggenheim Museum - CCBA 2019) and the projects *Software-based Art Preservation* (Ensom et al. 2021) and *Preserving Immersive Media* (McConchie et al. 2021) at Tate in London and have organized dedicated symposia (Smithsonian - TBMA Symposia 2014, AIC - TechFocus III 2015). Cross-institutional networks have been formed to transfer knowledge across disciplines, such as the *Community of Practice on Software-based Art* (Dekker and Falcao 2017) in the *Pericles* project and the *Software Preservation Network* (SPN 2022). In-depth case studies are used to map artwork needs and catalyze new conservation responses (Lintermann et al. 2013, Fino-Radin 2016, Phillips et al. 2018, Roeck et al. 2019, Sherring et al. 2021). The range of research topics is continuously expanding and covers, among many other topics: approaches to analysis for their examination and treatment (Engel and Wharton 2014, Engel and Wharton 2015, Ensom 2018); the continued access to legacy artworks through emulation (Rechert et al. 2016, Roeck 2018), EaaS - emulation as a service (Espenschied et al. 2013), virtualization (Falcao et al. 2014, LIMA 2018); collecting and conserving web-based artworks (Hellar 2015, Dekker 2018); the development of tools such as Webrecorder and Conifer by Rhizome (Webrecorder (software) n.d.; Rhizome - Conifer (software) n.d.), as well as Rhizome's research into the use of linked open data to describe web-based artworks and preservation information (Rossenova 2020b, Fauconnier 2018); the documentation of software-based artworks (Lurk 2013, Barok et al. 2019, Ensom 2019), the collection, preservation and treatment of source code (Swalwell and DeVries 2013; Di Cosmo et al. 2017; Engel and Phillips 2018); and the application of conservation ethics to the examination and treatment of software-based art (Engel and Phillips 2019).

In addition, notable artists have taken the initiative to reflect and advise on collecting and sustaining software-based works (McGarrigle 2015, Lozano-Hemmer 2015, Reas 2016) and researchers such as Qinyue Liu and Colin Post have also reviewed conservation practices by artists (Liu 2020; Post 2017).

Chapter 22 acknowledges the point of acquisition as a decisive moment in the life of a software-based artwork and guides the reader through the collection process (Section 22.2). Section 22.3 offers an in-depth look at the technical make-up of software-based art, and Section 22.4 explores a variety of conservation treatment practices.

GLOSSARY:

algorithm: a defined sequence of steps or instructions to accomplish a specific task.

API (Application Programming Interface): An API defines interactions between software applications; it is commonly used to exchange files or data.

bus: A communication system for transferring data between components of a computer, including wiring, connectors, ports, and protocols.

client-side: When application code runs on the user's machine, that is considered "client-side" e.g., the HTML/CSS code of a website. (See *server-side* for comparison.)

command line interface (CLI): This interface allows users to issue commands and run programs by typing text and using the keyboard for navigation. (See *graphical user interface* for comparison.)

comments (in source code): Comments are annotations, instructions, documentation, or other notes written within the code of a computer program which are meant for human readers and ignored by the computer.

CPU: Central Processing Unit; an integrated circuit which can execute instructions represented as computer code.

cross-browser compatibility: This is a feature of websites which run successfully under different browsers and fail gracefully when necessary.

downwardly compatible: Downwardly compatible software (also referred to as *backwards compatible software*) refers to a programming language or application that can continue to run files written for earlier versions of that language or software.

device driver: Also simply called a driver, a device driver is a computer program that is used to manage external devices such as a printer, scanner, or other equipment. Drivers are specific to both the computer's operating system and to the device.

emulation: The use of software or hardware tools called emulators, which allow one computer system to behave as if it were a different computer system, to run software on computer hardware which it was not designed for.

EPROM: Erasable Programmable Read Only Memory; non-volatile storage which can be read from, written to, and erased using a strong ultraviolet light.

EEPROM: Electrically Erasable Programmable Read Only Memory; non-volatile storage which can be read from, written to, and erased using an electrical signal.

executable: An application file that runs when accessed; for example, when one calls it up at the command line and hits <ENTER> or "double-clicks" on it with a mouse within a graphical user interface.

GUI (graphical user interface): A GUI offers the user graphical icons and audio along with devices such as a mouse. (See *command-line interface* for comparison.)

hardware environment: a set of interconnected hardware components which a specific software program requires to run. (See *software environment* for comparison.)

HDD: Hard Disk Drive; a non-volatile storage device which reads and writes data from a magnetic platter.

human-readable file or data: These contents can be opened in a text editor or at the command line and rendered without special characters so that a human being can naturally read them. For example, the digits directly below a bar code are the human-readable portion of the information.

integrated circuit (IC): A circuit contained on a single piece of semiconductor material. Also known as a microchip or chip.

library: A software library contains additional code that is used with a given application or programming language. In many cases, the library is considered a "third party library", meaning that the external library was not written by the same developers who wrote the application or programming language that calls the library.

OS: This acronym refers to "operating system", the software that controls a device's basic functions such as file management, managing peripherals, the user interface and more. Examples include *Windows*, *MacOS*, *Linux* and more.

PCB: Printed-circuit-board; a fabricated circuit consisting of a conductive material in a non-conductive substrate, that forms a base for mounting electronic components.

PROM: Programmable Read Only Memory; non-volatile storage which can be read from but only written to once.

RAM: Random Access Memory; volatile storage used to store data and programs when they are accessed.

ROM: Read Only Memory; non-volatile storage which can only be read from and not written to.

server-side: When application code runs on the server e.g., in PHP or Perl, it is considered "server-side". (See *client-side* for comparison.)

software environment: a set of interconnected software components which a specific software program requires to run. (See *hardware environment* for comparison.)

software obsolescence: As hardware, operating systems and other elements of a system change over time, many programming languages and software applications will not run as expected or will no longer run at all.

source code: Human-readable set of instructions, written in a formal programming language, that tell a computer what to do.

SQL (Structured Query Language): Often pronounced "Sequel", SQL is a language used to manage data and databases.

SSD: Solid State Drive; a non-volatile storage device which reads and writes data from solid state flash memory.

VCS (version control software): VCS refers to software applications that are used to manage and document change across computer programs, documents, and other files. *Git* (often used with tools by GitHub) is an example of a popular VCS application.

WYSIWYG (what you see is what you get): An acronym that signals when output to the printer or web reflects what the user sees on the screen e.g., when developing content using desktop publishing, web development or graphics software.

22.2 Collecting Software- and Computer-based Art

When software- and computer-based artworks are collected, their sustainability comes into focus. The responsibility for their care and maintenance is transmitted from the artist to the new owner, and each artwork's longevity will critically depend on the actions taken - or not taken - by its new custodians. Before a collected artwork breaks down or requires repair and maintenance, there are important steps that should already occur in the acquisition process to enable future preservation. Relevant information about the artwork and its underlying technical systems has to be gathered, preservation-critical components have to be collected, and rights and responsibilities cleared. Conducting an initial preservation risk assessment and projecting future needs of an artwork can critically inform the acquisition process and help to successfully sustain the artwork's collection life (Falcao 2019). This is why it is best practice to include conservators and other relevant stakeholders in the acquisition process as early as possible and to proactively define the deliverables needed to support the longevity of a work.

When bringing software-based artworks into a collection, it is ideal to examine and document them as soon as possible after their creation when they are still fully functional in their native *hardware* and *software environments* and display all of the behaviors intended by the artist. Surveying artworks later, perhaps years after their entry into the collection, bears the risk of software or hardware malfunction that could go unnoticed. As an example, when viewing the Guggenheim's web artwork *Brandon* (1998-1999) by Shu Lea Cheang prior to its 2016-2017 restoration, words that were intended to blink across the sprawling website appeared static. The words were still legible, and it would not occur to an audience unfamiliar with the work that the blinking function was indeed missing, as contemporary browsers no longer supported the HTML blink tags. Only source code analysis (see 22.3.6) revealed the missing functionality (Phillips et al. 2017; Engel et al. 2018). Had there been a simple video of the piece, recorded around the time of its creation, collection caretakers would have had a valuable reference when visually assessing loss and damages many years later.

Learning from experiences like this, the conservation community has developed a range of acquisition practices to support the future preservation of software- and computer-based works.

22.2.1 What is the Artwork? Understanding the Intended Experience

The best way to start an acquisition process is to experience an artwork first-hand, installed and running. This creates an immediate impression of its intended behaviors and potential implications for conservation and maintenance and critically informs the acquisition process. Many artwork features and behaviors may not be spelled out in the artist-provided information and are not easily transmitted through documentation. This is particularly true for interactive works that require audience engagement or data input.

When first experiencing a software-based artwork in operation, collection caretakers aim to answer questions like the following:

- What is the artwork intended to do and how is the audience expected to interact with it?
- What are the conditions of display and space?
- What are the employed technologies (software and hardware), and do they have an aesthetic or conceptual relevance for the artwork that could make them hard to replace?
- What could challenge the maintenance, upkeep, or repair of the work for the next few exhibitions, as well as far into the future?

Experiencing the work pre-acquisition, even if only partially installed during an acquisition committee meeting or in a gallery set-up, may be the last opportunity to gather relevant information before the new owner is charged with the next re-installation. It is therefore advisable to produce as much documentation as possible on this occasion, whether by means of video, via screen recording or a combination of different documentation methods (see Chapter 11).

22.2.2 Researching the Meaning, Making, History and Context

Contextual and technical research during the acquisition phase is a key preservation step for software-based art and lays the groundwork for appropriate conservation strategies. While variability and change are integral to all time-based media, they are particularly fast paced and urgent in software- and computer-based artworks. Any operating system (OS) or browser update, hardware failure, software obsolescence or conceptual preference (for instance, if an artist wants to see their historical web artwork updated to run on mobile devices) can be a driving factor for changes to an artwork. The research phase aims to answer questions like the following:

- What is the artwork's previous history of exhibition and change? How has it evolved or been updated over time?
- Were some iterations considered more successful than others, and if so, why?
- How was the artwork conceived and produced (technically and conceptually) and what is significant about it?
- Who was involved in the development of the work?
- Which programming language(s) or software technologies did the artist (or programmer) use, and for what reason?

- How do they relate to other hardware or software technologies that the artist/technician uses? Is this platform used regularly by the artist, or is it a one-off?
- Where does the artwork sit within the practice of the artist and the overall artistic and technological practice of its period?
- How common was that technology at the moment when the artwork was produced?
- How is the artwork experienced and perceived by the audience? the press? the public?

The information sources consulted during this research phase may include, among others: artist-provided manuals, relevant internal and external email correspondence, conversation notes, video recordings of the functioning work, images and videos of a work on social media, entries on media art databases (e.g., ADA | Archive of Digital Art (website) n.d.), catalogue texts, curatorial descriptions of a work, artists' websites, or GitHub pages. Gathering and organizing this information, e.g., with the help of templates (Guggenheim Museum - CCBA n.d.; Ensom et al. 2021; The Metropolitan Museum of Art - Sample Documentation and Templates n.d.), will support an informed discussion with the artist. Conducting formal acquisition interviews with living artists is a common practice in many contemporary art collections (see Chapter 17); in the case of software-based artworks, such interviews may also include programmer(s) and others with in-depth knowledge of the technological make-up of a work.

The process of analyzing and documenting artwork components is iterative and a deeper analysis of the system, as detailed under 22.3, will produce further information. Eventually, the object file of a software-based artwork will contain a range of documentation, including but not limited to a component-based inventory (see 22.2.3), display and configuration instructions, identity, and iteration reports (see Chapter 11), a component-based risk assessment (see 22.2.6 and Table 22.1), any source code documentation (see 22.3.6), and metadata in the collection management system.

Considering the context of the collecting institution, including the type and size of the collection, resources, and expertise available, is fundamental to devising sustainable preservation strategies.

22.2.3 Identifying the Anatomy of the Work

To effectively care for a software-based artwork, collection caretakers need to identify its interconnected hardware and software components and understand their relative importance in realizing the artwork's work-defining properties, as well as how each component potentially affects the way the system behaves.

The materials supplied by the artist will often consist of an executable file or sets of files and directories, a computer with peripherals ready to be set up in a gallery for display or, for web-based art, a group of files and folders to be uploaded to an institutional web server. Some artists may also supply source code, or other source materials. Additional components may include sculptural elements, temporary exhibition hardware (e.g., a projection screen or monitor), replacement parts or hard drives with back-up files and software not included in the OS.

For complex works with many elements it is helpful to start with a basic inventory listing all the known components of the artwork. The example in Table 22.1 illustrates how a component-based inventory can then be expanded to structure the analysis and risk assessment process and support the development of preservation strategies.

The inventory should record basic information on hardware, software, and data components, and note their functionality and the relationships among the components. For hardware components, peripherals and connectors should be considered, and standards, protocols, makes and models specified. See 22.3.1 for an overview of commonly encountered computer components and 22.3.2 for guidance on analyzing and documenting hardware. For an introduction to software technologies, see 22.3.4 through 22.3.8.

Table 22.1 Example of an initial, basic inventory and component-based risk assessment. This table models the type of information that may be captured in the process; the level of available detail, including the work-defining properties of computer components, may deepen with continued examination and stakeholder involvement.

Basic Inventory				**Risk Assessment**			**Preservation Strategies**
Components (hardware, software, data, project files etc.)	Make and Model	Function, specs, relationships	Work-defining properties	Significance	Hazards / Risk	Recoverability	
Computer	Torch RedFoCS (2003)	Executes artwork; is joint with monitor and mounted / hidden behind it; see spec sheet	Fits behind monitor; capable of running the OS and executable file, sends display information to monitor	Functional; conceptual; historical (early example of monitor with built-in computer)	Wear and tear; most at risk: hard disk, CPU, power supply, video cards and capacitors. Risk is high	Replaceable, as long as it remains invisible behind monitor and has appropriate video output	Controlled storage (climate, dust-free, ESD-safe); create and store disk image; source and set up spare computer (for possible replacement); clean regularly, particularly when on display
Monitor	Torch RedFoCS / LCD NEC LC (2003)	Displays artwork; joint with computer; see spec sheet	4:3 aspect ratio; slim black frame; screen size 17” to 19”; LCD panel type	Functional; aesthetic; conceptual (“discrete object”; 4:3 content)	Wear and tear; change of brightness and color balance; permanent screen lines. Risk is High	Replaceable with different monitor (17” to 19”, 4:3, slim black frame)	Controlled storage (climate, dust-free, ESD-safe); source back-up monitor (for possible replacement); clean regularly, particularly when on display

Video Connector Cable	VGA connecti on from PC to Monitor	Transmits image signal	Short, hidden behind monitor; compatible with computer output and monitor input	Functional (connection type not significant)	Wear and tear/ Low Risk	Replaceable with other hardware-compatible cables / connector types	No action needed (VGA cables are still available)
Operating System	Window s XP (home edition)	Runs artwork executable (Director .exe)	Capability to run artwork .exe	Functional	Obsolete; no support / security updates, but artwork is offline / Low Risk	Replaceable with OS that can run artwork .exe; or XP can be run on emulation	Store back-up copy of Windows XP in digital repository
Application ("the Artwork")	executab le file (.exe) created in Macrom edia Director 8.0	machine-readable instruction suited for OS Windows XP	Contains instructions and data dictating artwork behaviors (animation speed, colors, movement, shapes etc.)	Functional; aesthetic; conceptual; historical	Dependency on OS Windows XP, no hardware or network dependencies. Moderate Risk	If it does not run on recent Windows OS (test), run on XP emulation.	Store back-up copy in digital repository
Artist's project file	Project file (.dir), Macrom edia	Production element, required for treatment and	Allows conservators / programmers to fully capture (and potentially	Preservatio n-critical element	Proprietary, obsolete, limited support. Dependent on	Run legacy Director in emulated legacy environment	Store project file in digital repository; obtain and back up a copy of legacy platform Macromedia Director

	Director 8.0	examinatio n	reconstruct) artwork behaviors and to update any settings		obsolete Director platform. High Risk	, if needed.	8.0

22.2.4 Artwork Inspection

During an acquisition process, software-based artworks should be inspected in the conservation lab to ensure that none of the needed deliverables are missing, that the work is fully functioning, and to record the behaviors in the artist-supplied or artist-approved environment. The examination and documentation of the hardware and software components may be undertaken by the conservator themselves, or– depending on their skill set and the level of engagement of the artist–through collaboration with the artist and/or programmers and other technical specialists.

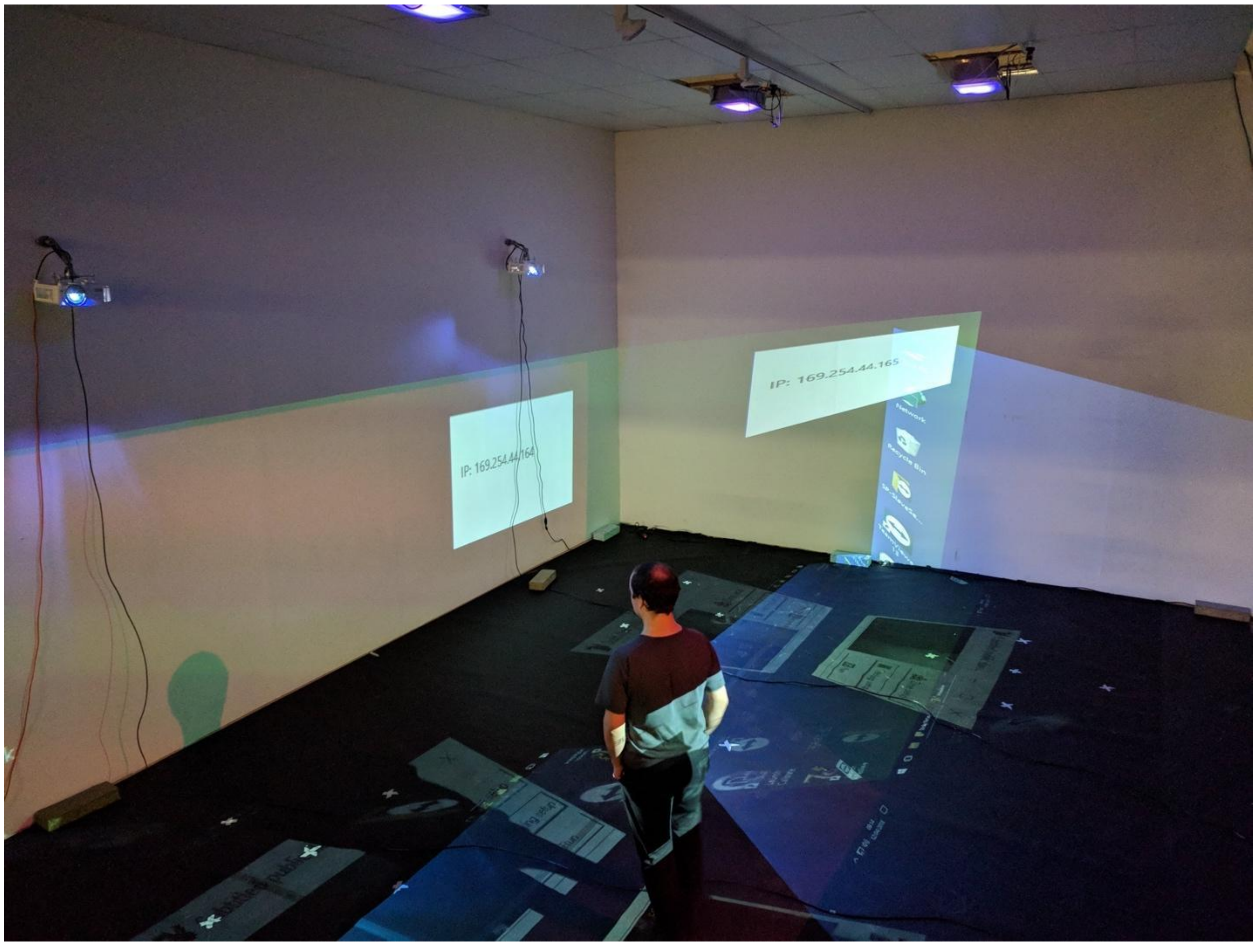
Creating an Inspection Environment

In order to test the software delivered, an inspection environment - made of suitable hardware and software - will be required. In some cases, this will be the hardware delivered by the artist. If no hardware is received with the artwork, the examiner has to set this up. In some cases, this environment will also be used for display. Ideally, the environment is created in collaboration with the artist or by using an artist-supplied specification. If this is not available, decisions can be made based on the format of the executable software (i.e., which operating system it will run on), the period in which the artwork was produced, and any knowledge of the technologies used in its production or previous displays. As the software environment is created through the process of installing and configuring an operating system and other programs, the components that were used to build it should be documented and archived.

In some cases, a desktop computer with the intended operating system, or even just a few browsers, may be enough for an initial inspection. However, some works--regardless of whether hardware was provided by the artist or not--may need to be fully installed to be correctly inspected. There may be limitations as to how much of the work can (or must) be installed for inspection purposes, namely for large works, or works that require a physical installation space with certain characteristics.

For example, to inspect and test *Subtitled Public* (2005), a software-based artwork in the Tate collection by Rafael Lozano-Hemmer, it is necessary to have a completely dark space with a minimum size of 8x8 meters, and with an appropriate floor covering. Testing this software can involve 3 or 4 people working for a week (see fig. 22.1).

<<INSERT FIGURE 22.1 HERE [T**BM_Chapter_22_Software_Figure_1_Lozano-Hemmer.jpg] Caption:**
Figure 22.1 Inspecting and testing the software-based artwork *Subtitled Public* (2005) by Rafael Lozano-Hemmer requires an 8x8m space and 3 to 4 people working for a week. Photo: Tom Ensom



Switching the Artist-provided Artwork on for the First Time

Before powering on a software-based artwork for the first time, it should be verified that an identical copy of the software and date that comprise the artwork is stored safely. Any computer or storage device can fail spontaneously at any time, potentially resulting in irretrievable data loss or corruption. Supplying power to or booting a computer can in itself cause it to fail if it is in poor condition. Risk increases with its age, hours of usage and improper storage conditions. A computer that has been sitting on a shelf for five or ten years could suffer a hardware failure upon being switched on (see 22.3.3). If a computer has been improperly stored, you may wish to ensure that internal components are fully dry and clean before use. Dust can be removed with a can of compressed air, an ESD-safe vacuum cleaner or equivalent, while ingrained dirt and debris can be removed with isopropyl alcohol and a lint-free cloth.

Especially if the software-based artwork runs on an artist-provided computer, a backup of the data contained on the computer (not just the files that make up the artwork) should be created. The ideal back-up is to create a disk image of the entire computer's storage devices, which includes all installed systems and applications. Chapter 14 provides a detailed overview and guidelines on how to create disk images. The data must be safely backed-up (see Chapters 8 and 13), and only then should the computer be turned on. At this point any other related hardware (e.g., peripherals such as cameras or joysticks) should be connected, so that the full functionality of a work can be tested and documented.

Before the artwork itself is run, it is advisable to use built-in system tools such as the system report to create basic documentation of the hardware and software environment (see 22.3.2). This way, the configuration is captured directly from the system, rather than derived from inspection or manuals, and before any operational failure may be triggered by executing the artwork.

Running and Inspecting the Software and Hardware

Once the system documentation is created, the software can be run. Especially when inspecting older, fragile artworks that have a higher risk of failure, this moment should be considered a valuable opportunity to create video documentation of the work running as well as of the user interface on the computer. When inspecting the software and hardware running, collection caretakers should aim to generate answers to questions like the following:

- Are the artist-supplied components (equipment, software, and data) complete and working correctly?
- Are we missing any crucial information necessary to run the piece, such as passwords?
- Which software is employed by the artwork, in which version, and configuration and configuration?
- What is the setup and calibration process required to run the software correctly?
- When running correctly, what is the software supposed to do functionally and aesthetically (what are its audiovisual, kinetic and/or haptic behaviors, including colors, image characteristics, movement, speed, duration, data input, interactivity etc.)?

Answering some of these questions may require an in-depth analysis of the artwork as described in 22.3.

Understanding the Work-defining Properties

The concept of “work-defining properties" was introduced by Laurenson (2006) and has since been widely adopted in the time-based media conservation community. It serves to identify the properties of the artwork — such as hardware characteristics, audiovisual content, installation components and spatial parameters— that define its identity, are essential for its uncompromised integrity and should be preserved. For software-based art, Laurenson (2014, 24) expanded these general properties to include the appearance (including sculptural hardware components), behaviors, and modes of interaction. In addition to these discernible characteristics, significance in software-based art has also been attributed to its original source code, namely the artist’s or programmer’s choice of technologies, programming languages and coding styles (Engel and Phillips 2019, 181).

Developing an understanding of the artwork's work-defining properties is a critical step in the risk assessment process. This can be supported by a wide range of information, including interviews with the artist (and other stakeholders), technical specifications, source code and documentation of past installations, an understanding of which will be informed by the conservator's own knowledge and experience. One approach to this is to identify work-defining properties at the component level (see Table 22.1). However, care should be taken to ensure that these are not interpreted as fixed or authoritative. Attaching significance can involve subjective judgements and a complete picture of the artwork's identity will continue to emerge throughout its life.

22.2.5 Conducting Risk Assessment for Preservation

Risk assessment supports the development of preservation strategies and helps to prevent damage and loss to objects and collections. The concepts of risk assessment and risk management are widely used in cultural heritage conservation as a way of identifying hazards, threats and risks for specific objects and collections as well as evaluating the impact of those risks and finding ways to mitigate them. The ICCROM - International Centre for the Study of the Preservation and Restoration of Cultural Property defines risk as “the chance of something happening that will have a negative impact on our objectives" (ICCROM 2016). Building on these practices, Agnes Brokerhof (2011) developed a risk assessment methodology for contemporary art installations, which also proved helpful for software-based art (Falcão 2010). Unlike traditional fine art conservation, where change is commonly seen as loss, technology-based artworks often must change to retain their functionality. Moreover, the possibility of full recovery, by which damaged or failing components can be replaced without change to the work-defining properties of an artwork, means that recoverability, or the balance between technical feasibility and affordability, becomes a key factor when identifying and prioritizing risks and devising preservation strategies.

The following section explains how to identify and evaluate risks for the conservation and display of a software-based artwork. In the process, each component’s significance for the artwork is assessed, including its functional, aesthetic, or conceptual significance (see Table 22.1). Then, its risks of failure and loss, and its potential recoverability from a conceptual and technical perspective are identified. A similar risk may impact different components in different ways, depending on their significance and ability to recover from failure.

This risk assessment is best reviewed and agreed upon with the artist(s), creator(s), curators, and other stakeholders to build consensus around preservation, maintenance, and display strategies.

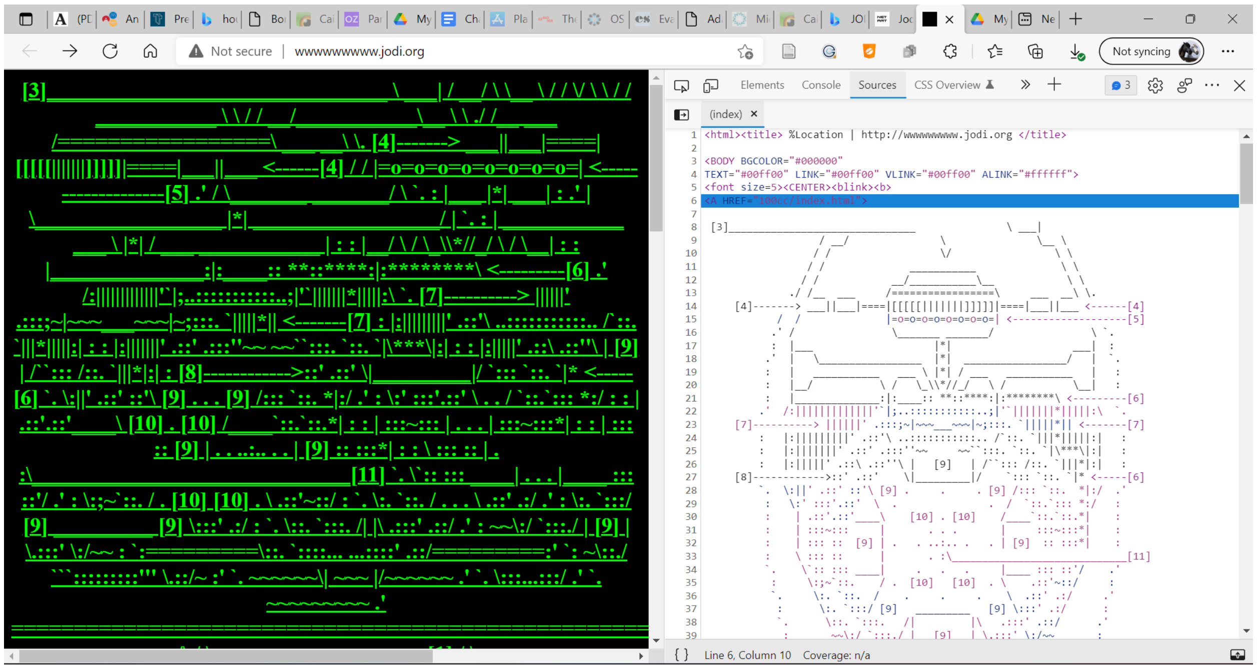
Understanding and Attributing Significance

‘Significance’ refers to “the values and meanings that items and collections have for people and communities" (Russel and Winkworth 2009) and reflects the value attributed by different stakeholders; a conservator and an artist may value a specific component differently. Moreover, attributed significance to the same component may vary in the context of different artworks: an ordinary-looking desktop computer may carry a purely functional significance for one artwork but be of conceptual significance in the context of another artwork, e.g., if the artist chooses to display it on the gallery floor. The computer could also carry historic value if it represents a significant point in art history, the history of technology, or a key moment in the artist's practice. Artwork components often combine several types of significance at once, in varying degrees: a computer could have a functional and a historic significance, but the functionality might be considered more significant for the artwork than its agency in representing a certain point in history. One way of attributing significance is through an understanding of the artwork's identity and work-defining properties.

Table 22.1 exemplifies how significance attribution (as an integral part of the risk assessment process) considers all hardware and software components, including monitors, cables, operating systems and associated networks and databases.

Especially when planning conservation treatments (see 22.4), it may be necessary to break down software and hardware components further, e.g., to consider the significance of a particular graphics card inside a computer, or the programming language underlying the software. For example, the HTML code in JODI's web artwork *wwwwwwwww.jodi.org %Location* (1995) does not only functionally create the page, but also carries conceptual and aesthetic significance, as it is written so that the HTML code takes the form of a drawing (fig. 22.2).

<<INSERT FIGURE 22.2 HERE [**TBM_Chapter_22_Software_Figure_2_Jodi.png**] Caption: **Figure 22.2** The webpage (left) and underlying HTML code (right) for the web artwork *wwwwwwwww.jodi.org %Location* (1995) by JODI. Screenshot: Patrícia Falcão


Not secure | wwwwwwwww.jodi.org
Not syncing
Elements
Console
Sources
CSS Overview
(index)
Line 6, Column 10 Coverage: n/a

```
<html><title> %Location | http://wwwwwwwww.jodi.org </title>

<BODY BGCOLOR="#000000"
TEXT="#00ff00" LINK="#00ff00" VLINK="#00ff00" ALINK="#ffffff">
<font size=5><CENTER><blink><b>
<A HREF="100cc/index.html">
```

The relative significance of artwork components is the basis for their subsequent assessment of recoverability. For instance, if a computer's significance is mostly functional, the spectrum of options for recovery–from careful storage to complete replacement–are available. If it is deemed to have aesthetic significance it may be replaced by other means, as long as the replacement is invisible. If the value of the computer is deemed to be conceptual, the options for preservation may be limited to storing the original computer, like a traditional sculpture–in extremis, to the detriment of its functionality.

Generally, in the context of time-based media art, high significance is attributed to functionality with the aim to preserve a functioning artwork, often overriding historical or aesthetic value of a component by allowing for its replacement. In other words, faced with a work that stopped functioning, many, but not all, artists will be happy to make changes to maintain that functionality, and often to technically improve it. The latter may raise ethical concerns, as these changes may compromise historical or aesthetic significances.

It is important to highlight that different collections, communities and other stakeholders may value different aspects of a work. For instance, early computer art, by artists such as Frieder Nake or Vera Molnar, has long been collected by museums only in its outputs as prints. The value of the underlying algorithms has been neglected in that context, but it was valued at the time of creation by the artists themselves and journals such as "Computer Graphics and Art" (Della Casa 2012), which chose to publish that code. Currently, the community around the Recode project is re-using that code and translating it to Processing (Recode Project, n.d.).

Identification of Hazards and Estimation of Associated Risks

The agents of deterioration for traditional objects of cultural heritage include physical forces, fire, water, criminals, pests, contaminants, light and UV radiation, incorrect temperature, incorrect relative humidity, and custodial neglect (Waller 1994). In spite of the misleading discourse about the immateriality of digital objects, all of these hazards are relevant and will apply to data carriers (even servers in “the cloud”), equipment, and any other physical components of a software-based artwork. Brokerhof (2011, 97) identified further tangible and intangible agents of deterioration for media installations: electricity; wear and tear; plastics and metals degradation; information dissociation; malfunctioning and misinterpretation. Further to these, the digital preservation community has established additional hazards specific to software, and more broadly digital objects, such as obsolescence, data loss, rendering issues or inaccessibility (Digital Preservation Coalition (DPC) 2022). Fino-Radin identified three primary risks faced by new media artworks in the Rhizome collection: diffusivity (i.e., a tendency to reference external data sources), data obsolescence, and physical degradation (Fino-Radin 2011).

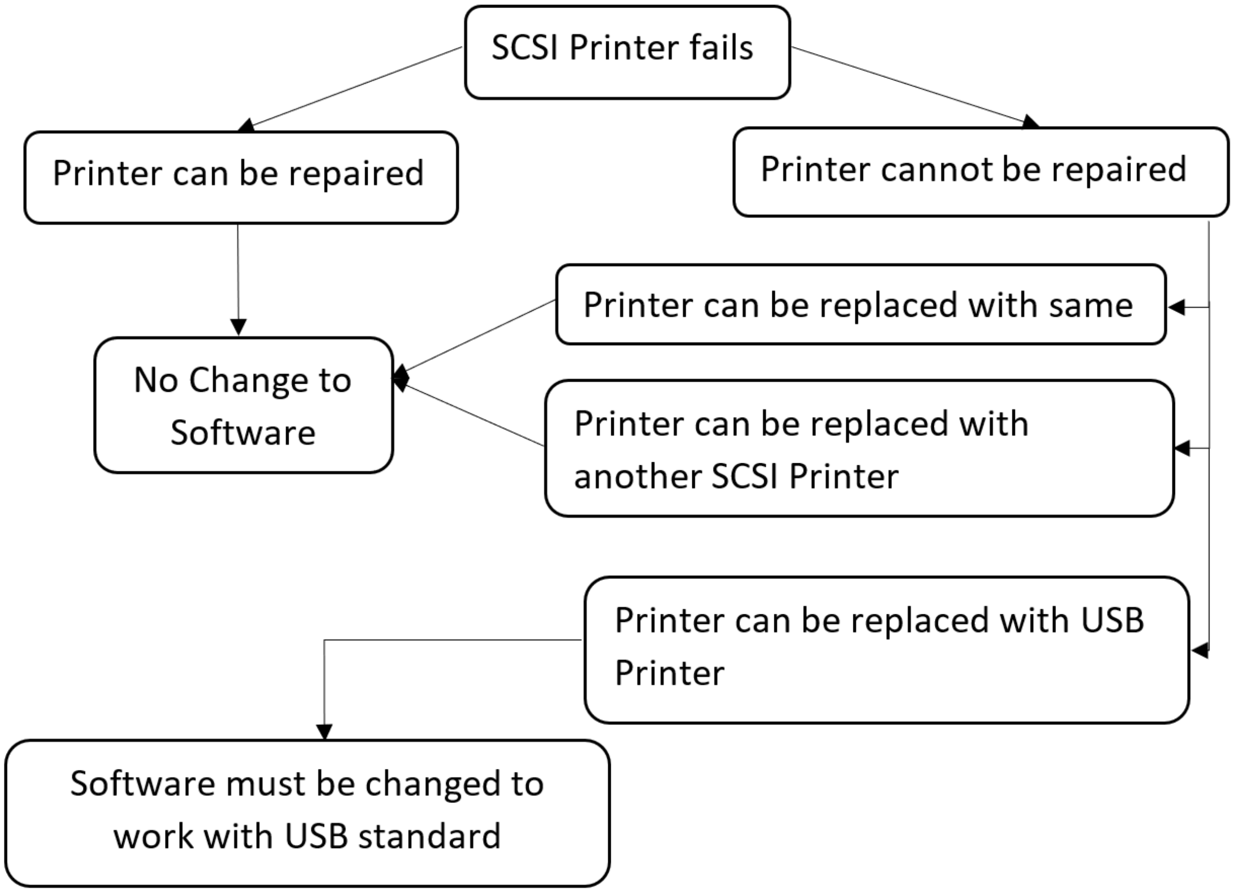
The magnitude of a specified risk depends on three factors: the likelihood of a hazard happening, the impact of that hazard, and the possibility of recovery. For example, the risk that a file is lost if it is stored on a single hard-drive is very high, as it is known that hard-drives fail, sooner or later. If copies of that file are stored on multiple storage devices, the risk of losing that file is much lower.

Assessing Recoverability and the Impact of Obsolescence

Even before hardware and software components fail, they may be affected by obsolescence, “a condition that affects an element or system when it or its parts are no longer commercially available or supported by the industry that initially produced them” (Falcão 2011). It is important to understand that hardware or software does not stop working because it is obsolete. The issue is that for replaceable components, it becomes increasingly difficult and expensive to repair or replace them once they are obsolete, and the increasing incompatibility of obsolete hardware and software with newer environments compromises their (correct) function. More staff time must be invested into sourcing replacements and dwindling specialty expertise is increasingly costly. For factors that drive hardware failure and obsolescence, see 22.3.3, and for software obsolescence, see 22.3.7. For an introduction to treatment responses to obsolescence, see 22.4.

A decision-making approach can support the process of assessing the recoverability of artwork components and understanding how obsolescence impacts their recovery in the context of their system. The example in fig. 22.3 explores the recoverability of an obsolete SCSI (pronounced "scuzzy") printer component of a software-based artwork, the different options for its recovery, and how after a certain point this may drive the need to change the whole system.

<<INSERT FIGURE 22.3 HERE
[**TBM_Chapter_22_Software_Figure_3_RecoveryDiagram.png]** Caption:
Figure 22.3 Recoverability assessment, here of an obsolete SCSI printer in the context of a software-based artwork. Diagram: Patrícia Falcão


SCSI Printer fails
Printer can be repaired
Printer cannot be repaired
Printer can be replaced with same
No Change to Software
Printer can be replaced with another SCSI Printer
Printer can be replaced with USB Printer
Software must be changed to work with USB standard

Assessing recoverability during the acquisition process helps to identify the deliverables needed to support the artwork's recoverability in the future. In this example, the need for spare SCSI printers and printer parts was identified as a recovery option until they can no longer be sourced.

Once printers with SCSI connections are no longer available, the software will need to be changed to output to USB printers, or the common connection at that point. Having the source code will make this intervention in the code much easier (see 22.4).

In summary, the main questions regarding recovery/replacement are:

- Is the element replaceable?
- Will its replacement lead to other changes to the system?
- Can an appropriate replacement be found or produced?
- Do we have the means (conservation elements, expertise, and funding) to produce that replacement?
- Is there a preferred approach if the element cannot be replaced? If the work cannot be recovered, is exhibiting it as documentation an option?

22.2.6 Collecting Preservation-relevant Components, Information and Rights

The sustained collection life of a software-based artwork critically depends on the hardware and software components, and on information and rights that the collector receives from the artist or gallery. Time-based media conservators play an active role in negotiating and obtaining deliverables. The collection team should establish the deliverables in close dialog with the artist/gallery and list them in the purchase agreement (see Chapter 17 Appendix). This way, the transaction is clearly defined for both parties, and funds can be released upon successful quality and condition control of the received deliverables.

Artist-provided Deliverables

Acquisition deliverables for software-based artworks depend on the type of artwork and its technological dependencies, the artist's preferences, a collection's resources, and the preservation awareness of the involved parties. In general, the collection goal is to enable the new owner to responsibly care for the work, to maintain and update it, to examine and treat it and thereby to manage its change without compromising its integrity–even if the artist is not or no longer available to consult. To support this transition of care and responsibility, the artist should be expected to supply:

- A full, running version of the work, including all hardware and software components. This set-up acts as an artist-approved reference of how the work should appear and function. If the artist does not supply all components, the collector should source or build the set-up and have it approved by the artist at the point of acquisition, or the first gallery display of the work.
- Technical specifications for hardware that is not supplied and should be sourced by the collection, including specifications on the future replacement of supplied hardware.
- Comprehensive installation instructions. This includes detailed instructions on how to install the artwork in a variety of spaces and conditions; how to install, configure and run the software; how to handle hardware and software maintenance and failures (troubleshooting, replacements, updates); and whom to contact if specialty expertise, service or supply is needed.

- Stockpile of hardware or replacement parts, e.g., in the case of rare legacy equipment or custom-made parts.
- A complete back-up copy of the software required to run the artwork, including executable files; software dependencies; firmware.
- Where executables can be produced for multiple operating systems, e.g., a Unity executable can be created for MacOS, Windows or Linux environments - an artist can be asked to produce executables for one or more operating systems.
- For web-based works, it is important to agree on who is responsible for hosting the website, and ideally the collector will have a full back-up of the site, even if not hosting the site themselves. The rights to any significant domain must also be acquired at this stage.
- The source code of the project, as well as file directories, text files, media assets, libraries, project files (e.g., for a development environment such as Flash, Director, Xcode or Unity, or 3D print files etc.).
- Granting of rights (via purchase agreement or non-exclusive license): the artist should grant the collector the right to show, publish and preserve the artwork; to create copies of the work for the purpose of preservation and access; and to treat the work, e.g., to migrate or update the code or project file (preferably in consultation with the artist, as long as the artist is available) to sustain the life of the artwork.

"Obtaining the uncompiled, human-readable source code in addition to the compiled, machine-readable executable file (...) is perhaps the most crucial act of preventive conservation." (Engel and Philips 2019,192). Section 22.3.6 shows how source code analysis is a critical tool for identifying artwork behaviors, especially randomized, generative, and other behaviors that cannot be quantified and qualified through human observation. Source code is also often needed in the long term to maintain, update, or migrate a software-based artwork (see 22.4.8), and to ascertain software and systems dependencies that the conservator must take into account for conservation and future treatment.

However, collectors may not always succeed in their negotiations to obtain the source code in addition to the executable file. While some preservation-conscious artists like Rafael Lozano-Hemmer, Jürg Lehni and Cory Arcangel share their code with collectors or GitHub communities, other artists feel more protective of it. Contractually agreeing on who has access to the source code, and for what purposes, can be very helpful in alleviating any concerns and obtaining an artist's consent. Not having access to the source code increases the risk of losing access to the artwork in the future. Although it does not preclude preservation, intervention may be significantly more technically challenging and costly without the source code, requiring reverse engineering of the software (see 22.3.6).

Sourcing and Creating additional Components as an Act of Preventive Conservation

Additional measures are recommended to be taken by the collector to broaden future preservation options:

- Create an identical exhibition copy of the whole artwork system, both hardware and software. This limits the use of the artist's original over extended periods of time, prolonging its life as a reference for the creation of new copies and systems.
- Create a disk image (see Chapter 14), either of the original computer or created from scratch by installing the artwork software on a disk image with the correct software environment. This supports access not only to the artwork software but to the whole software environment that surrounds it.
- Gather supporting software in its artwork-relevant versions, including production software (e.g., Adobe Flash, Visual Basic, Max MSP, Processing), media playback software (e.g., RealPlayer, Adobe Flash plug-in), operating systems, libraries, plug-ins, and drivers. In some cases, the software may not be available to purchase any more and will have to be sourced through the second-hand market.
- Record details of license keys and/or account information required to access or update these components, where applicable.
- Extract and store media files separately outside of the software (e.g., image or sound files animated on a Director timeline) to reduce the risk of access loss and to enable future reconstruction of the artwork with different technologies.
- If the collector is unable to obtain the source code, future examination and treatment may depend on decompiling the artist-provided executable. Even if this path does not guarantee success (Engel and Phillips 2019, 192), the collector should source the appropriate decompiler and compiler software immediately as an act of preventive conservation. The compiler software is not artwork-specific but may be difficult to obtain in the future when it might become necessary to recompile the artwork.
- Create comprehensive audiovisual documentation such as a video recording, narrated screen recording or similar to capture the artwork's intended appearance, experience, behaviors, and functions.

If other preservation-relevant deliverables are not received, the collector is well advised to step in and source or create these components on the artist's behalf (and seeking artist approval, where applicable), including stockpiling equipment and replacement parts or sourcing software the artwork depends on. For handling proprietary software in the context of preservation, the Software Preservation Network published a report on how fair use may be applied to software preservation (ARL 2019).

22.3 Understanding Your Software-based Artwork

In order to effectively care for a software-based artwork, conservators will need to be able to identify the technologies used, make sense of their role within the system of the artwork, and understand how these technologies are likely to be impacted by the passing of time. Through a process of analyzing and documenting their findings, conservators should aim to capture critical information to guide preventive conservation now and treatments in the future. In this section the authors aim to support the in-depth examination process by providing an overview of commonly encountered hardware and software technologies, their dependencies, and implications for preservation. Key failure and obsolescence risks are explored, and practical guidance on approaches to analysis and documentation are provided.

Software-based artwork systems are made up of two types of components: *hardware* and *software*. Hardware components are electronic devices which are capable of processing and responding to instructions (or other inputs) and generating tangible outputs (such as images on a computer monitor). The hardware associated with a software-based artwork will usually consist of at least a digital computer containing a *microprocessor*. This is an *integrated circuit* — a circuit contained on a single piece of semiconductor material (also known as a chip) — capable of processing binary data through logical and mathematical operations. Software components consist of encoded instructions and associated data that tell a computer what to do. These instructions are written in programming languages and can be stored and run by the computer on demand.

22.3.1 What is Hardware?

Hardware consists of the physical, electronic components of a computer system. Modern computer systems make use of an array of hardware components, which together with software components, turn instructions and other inputs into tangible outputs. The purpose of a system will dictate the kinds of hardware components used. For example, a computer used to run virtual reality software will need to incorporate powerful graphics processing hardware and attach to specialized peripherals, while a computer that can fit inside a smartphone will need to use components of a reduced size and consequently, with reduced performance.

In many cases, the hardware used to realize a software-based artwork may include use of display equipment common to other forms of time-based media. For example, an artwork may utilize monitors or digital projectors to display moving images alongside audio equipment to reproduce sound. These kinds of hardware are sometimes referred to as *peripheral* devices in the context of computing. The primary focus of this section is on computers, as they are essential to the execution of the software and thus central to the realization of the artwork.

<<INSERT FIGURE 22.4 HERE
[**TBM_Chapter_22_Software_Figure_4_CompFormFactors.png]** Caption:
Figure 22.4 Various types of computers. Clockwise from top left: iMac G3; custom desktop tower; Raspberry Pi Model A single-board computer; Arduino Uno Rev3 microcontroller; off-the-shelf COMPAQ desktop; Mac Mini. Photo: Tom Ensom

Conservators are likely to encounter a diverse array of computer hardware when caring for software-based art. In this section we introduce common types of computer hardware and consider why and how they are used by artists. Particular preservation risks to hardware are also highlighted, but it is important to note that all hardware is at risk of loss in the long-term (see 22.3.3).

Common Computer Components

Computers are made up of electronic circuits, and can employ a single circuit board, or many circuit boards connected via suitable interfaces. Desktop computers are typically made up of many removable components connected to the motherboard via internal (concealed within the computer case) and external (accessible on the exterior of the case) *buses*. This approach means that individual components can be replaced or upgraded without having to replace the entire system. As computer size decreases, there is a higher likelihood of components being soldered directly onto a reduced number of circuit boards. Microcontrollers represent this at the smallest extreme, where all of the hardware components required for a particular function can be contained on a single circuit board, or even a single integrated circuit. This offers much reduced size but makes replacing or repairing individual components more challenging and alters prospects for maintenance in the course of conservation treatment (see Section 22.4).

Despite the wide variety of uses of computers and the myriad forms they take, there are certain common components found across modern computers and which are frequently encountered when caring for software-based art. These are summarized in Table 22.2, as a reference for the use of this terminology in subsequent sections. While the components listed are framed in the context of separate devices contained in a desktop computer, many of the components listed can also be found built into a single circuit board or integrated circuit.

Table 22.2 Description of common computer components and the key specification information to capture when documenting their specification.

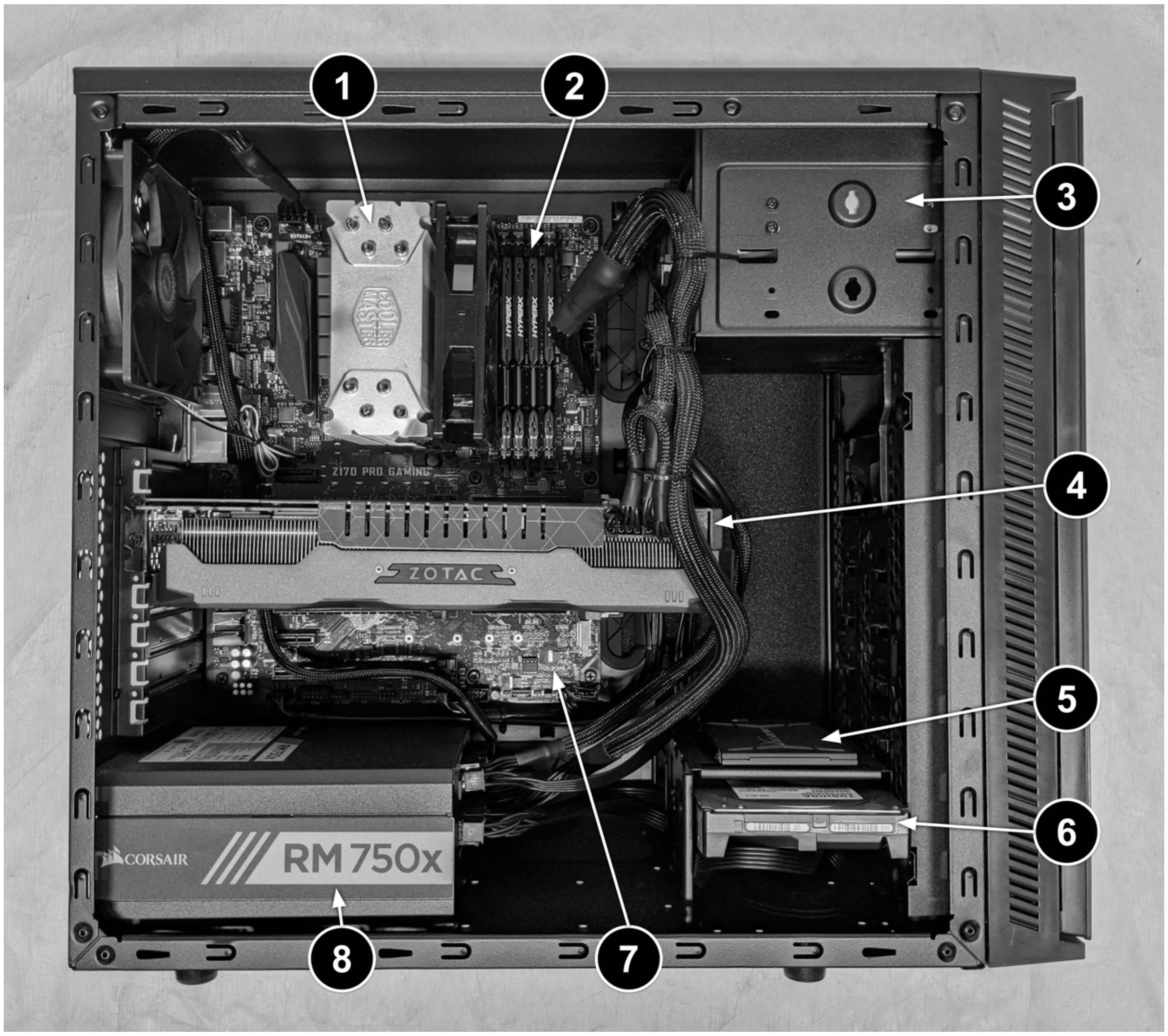
Component	**Description**	**Key specification information**
Motherboard	The motherboard is the circuit board to which the computer's other hardware components connect and through which they communicate. Also known as a Mainboard or Logic Board.	Model Supported interfaces
Central Processing Unit (CPU)	The CPU carries out operations based on instructions written in computer code. A CPU implements a specific instruction set architecture (ISA) which defines the kind of instructions that can be executed, e.g., x86-64 in modern desktop computing, or ARM64 in mobile computing.	Model Architecture Core number Clock speed
Random Access Memory (RAM)	RAM is volatile memory to which programs and data are loaded when they are executed or used, allowing for rapid access by the CPU. The information in RAM is lost when the machine is turned off or re-booted.	Model Capacity
Storage Devices	Devices for storing and accessing digital data. At least one storage device within a computer will contain a software environment consisting of an operating system, software programs and user data.	Model Capacity Host connection interface (e.g., PATA/IDE, SATA, SCSI)
Graphics Processing Unit (GPU)	A GPU is a device for rendering 3D graphics. While sometimes taking the form of standalone expansion cards,	Model VRAM (Video RAM) Host connection, if removable

	GPU functionality can also be integrated into a CPU.	(e.g., AGP, PCIe)
Network Devices	Devices which support or extend the computer's networking capabilities.	Model Host connection (e.g., AGP, PCIe) External interfaces (e.g., 8P8C ethernet)
Audio Devices	Devices which process, route and/or generate audio signals such as sound cards or audio interfaces.	Model Host connection (e.g., PCIe, FireWire 400, USB 3.0) Supported input and output connections (e.g., 3.5mm jack, RCA)
Power Supply Unit (PSU)	Device which routes power to components that require it.	Model Wattage
Peripheral Devices	Removable devices which extend the functionality of the computer system such as cameras, sensors, human interface devices.	Host connection interface (e.g., USB 3.0, FireWire 400)

Desktop and Laptop Computers

Computers can be manufactured to conform to a variety of *form factors*: a term for characterizing different size and shape profiles. Full-size desktop computers incorporate discrete hardware components in a relatively large case and, as the word desktop implies, are intended for home and office use. They contain full size, off-the-shelf hardware components (which are usually easily accessible via a removal panel) and therefore tend to be the most powerful and favored for high performance applications, such as real-time 3D rendering. The components of an example desktop computer are annotated in Figure 22.5 below. Desktop computers can be bought prebuilt or built to a custom specification using off-the-shelf components either by the artist or a third-party supplier. In the event that an internal component fails, it can usually be removed and replaced with a similar component (if available).

<<INSERT FIGURE 22.5 HERE [**TBM_Chapter_22_Software_Figure_5_CompComponentsAnnotated.png]** Caption: **Figure 22.5** The side panel of this desktop computer has been removed to reveal the internal components: 1) Heatsink and fan array mounted above the CPU; 2) Four sticks of RAM; 3) Optical drive mounted in a 5.25” drive bay; 4) GPU attached to a PCIe bus on the motherboard; 5) 2.5” Solid State Drive; 6) 3.5” Hard Disk Drive; 7) Motherboard; 8) PSU. Photo: Tom Ensom



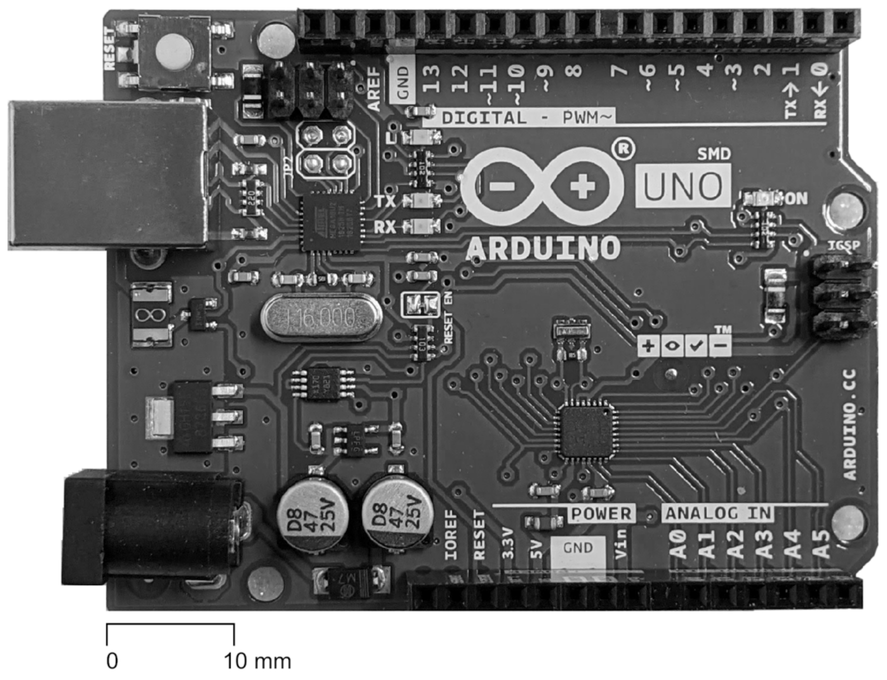
Small form factor desktops contain similar components to full-size desktops, but with a much-reduced physical size. Their smaller size sometimes comes at the cost of maintainability, as components can be difficult to access or in some cases are soldered directly onto the motherboard. They are typically sold as mass-produced, pre-built units, such as the Mac Mini or Intel NUC ranges. While they may be less performant than full-size desktop computers, their smaller size means that they can be more easily concealed in an exhibition space. Laptops often use similar components to those found in small form factor desktop computers, while also integrating essential peripherals into the case such as a screen, keyboard, and trackpad. They can suffer from similarly reduced maintainability.

Microcontrollers

Microcontrollers are small computers contained on a single microchip (an integrated circuit or IC). A microcontroller IC will contain at least a CPU but may also incorporate a small amount of RAM and persistent storage. The latter may be programmable and modifiable to some extent, depending on whether it uses ROM, PROM, EPROM, EEPROM or flash memory. Unlike a desktop computer, a microcontroller does not run an operating system, so it can only be programmed by connecting it to another computer. Microcontrollers can carry out a reduced set of possible operations (described in the datasheet for that model) in comparison to a microprocessor-based, general-purpose computer. For example, a microcontroller might be used to control the movement of a servo motor or an array of lights. Microcontrollers are likely to be chosen by an artist where a low cost and small size computer is needed to implement a simple control system.

A microcontroller IC needs to be built into a circuit board with other components to be used. To simplify the process of working with a microcontroller, circuit boards containing a microcontroller IC and other components are manufactured and sold as units (for example, the Arduino (see fig. 22.6) and Basic Stamp series. These are known as *single-board microcontrollers*. These units make use of an integrated interface (e.g., USB, RS-232) and development software to simplify the process of loading instructions onto the device. This contrasts with a standalone microcontroller IC, where compiled code must be loaded using a specialized device — known as a *programmer* — connected to a host computer. The latter is rarely used by artists today due to the specialist requirements for customization in comparison to single-board microcontrollers but may be encountered where an artist has worked with an engineer, particularly in older artworks before the wider availability of single-board microcontrollers.

<<INSERT FIGURE 22.6 HERE [
TBM_Chapter_22_Software_Figure_6_ArduinoUno.png] Caption:
Figure 22.6 Arduino Uno Rev3 SMD microcontroller. This model uses an integrated ATMega328P microcontroller IC and USB connection. Photo: Tom Ensom



Arduino, microcontrollers, and related technologies can play two roles in a conservators' work: as artistic medium of artworks that enter a collection and need to be conserved; and as a conservation strategy when migrating other works to this technology for preservation purposes.

On microcontrollers:

sasha arden: Microcontrollers can play two roles in a TBM conservators' work. I was trying to think of all of the artworks I have encountered or know of that definitely use microcontrollers. And I came up with just as many conservation projects that I've been involved in or know of that use microcontrollers as a treatment. So, it's about half and half.

Mark Hellar: I would say that each artwork is different, right? So, some artworks have microcontrollers that talk to the internet, some have microcontrollers that use Bluetooth and there's a vast array of sensors. So, we could say, "Oh, it's an artwork with a microcontroller", but I think it might make more sense to say, "What does this artwork do? And what are the significant properties?" And then I could take it out of this paradigm that this is a microcontroller-controlled work and, I'd spend time with the artwork to really understand what we would need to do and what elements we would need to pursue to care for this artwork.

sasha arden: It is wise to document one's own treatments using micro-controllers as well as documenting artworks that use them. I record the hardware that I use, the make and model of the microcontroller and I create a circuit diagram and list all of the electronic components that were in the external circuit. I make wiring diagrams of how all those things were connected and create a text file of the program that I wrote, and I also take a photo as a kind of a screenshot, when I can.

Mark Hellar: There are a number of tools to help with this process. They're known as "PCB tools" which means "printed circuit board design tools". What you do is you create a schematic and there are standard symbols for electronic schematics. Then you can convert them so that you can lay out a printed circuit board and have that refabricated, for example to help you to migrate from one microcontroller to another.

sasha arden: Also, even just having a standard electrical schematic would be enough for someone who can read that to build a new circuit by hand.

Mark Hellar: In terms of proprietary and open-source microcontrollers, you need to consider, "What does that mean in a hardware context?" Keep in mind that the Arduino project publishes all of the schematics and the source code so you could just go out and get some parts and put together your own Arduino on a breadboard and you can just download the source code and program the bootloader for it. But that is clearly not true for an iPhone! *(laughter)*

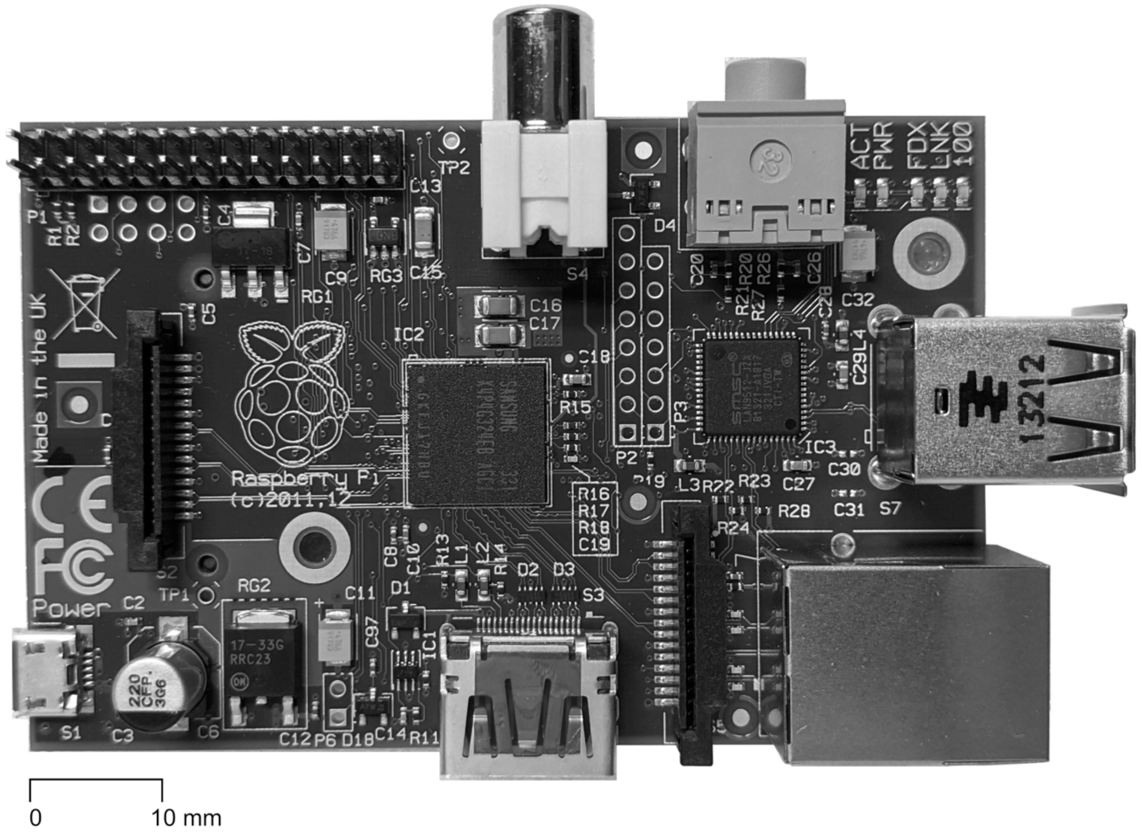
From interview with sasha arden and Mark Hellar, October 26, 2020 (a rden and Hellar 2020)

Microcontrollers should be well-documented to ensure their function is understood and could be recreated. Care should be taken to ensure that any code stored on the IC or board is also stored and preserved separately, as it is not possible to create disk images from microcontrollers and it may be difficult to retrieve data from them at all once it has been transferred. Microcontrollers are also harder to maintain than desktop computers, as components are usually soldered directly to the circuit board, making replacement difficult.

Single-Board Computers

Single-board computers incorporate a set of discrete hardware components on a single circuit board, but in contrast to a single-board microcontroller include a standalone microprocessor rather than the microcontroller IC. While they can look superficially similar to single board microcontrollers, they are actually closer to desktop computers in terms of purpose. The increased sophistication of the components in comparison to a microcontroller means that they are suitable for use as general purpose computers, can be easily connected to computer peripherals (such as a monitor and keyboard) and can run desktop operating systems. Single-board computers are found inside smartphones and media players. They are also available as standalone devices, such as the Raspberry Pi series (see fig. 22.7). These may be used by artists where the complexity of their software exceeds that which could be supported by a microcontroller (such as running an operating system or using a display output) but a similarly reduced size or lower cost is required.

<<INSERT FIGURE 22.7 HERE [**TBM_Chapter_22_Software_Figure_7_RaspberryPi.png]**
Caption:
Figure 22.7 Raspberry Pi Model A Revision 2.0 single-board computer. This model uses a Broadcom BCM2835 ARM CPU and includes significantly more input/output options than a single-board microcontroller, including HDMI, composite video, 2x USB 2.0 ports, ethernet and 3.5mm audio. Photo: Tom Ensom



As for microcontrollers, single-board computers are difficult to maintain as components are soldered directly onto the circuit board and are not easily replaceable. The recoverability of code and other stored data is highly dependent on the device and nature of the storage medium. Data may be retrievable where it is contained on removable storage media (e.g., an SD card or USB stick), but storage integrated into the board may be very difficult to access.

Peripherals

The capabilities of a computer may be extended by attaching peripherals, easily removable hardware devices which extend the functionality of the host system by providing additional inputs or outputs. Examples of frequently encountered peripherals in software-based art are monitors, cameras, sensors, networking devices and human interface devices (such as keyboards, mice, and controllers). Each will have its own unique characteristics and understanding its importance within the larger system will have to be approached on a case-by-case basis. It is generally important to identify the interface used to connect the peripheral (e.g., RS-232, Firewire, USB) as the obsolescence of these interfaces is a common source of long-term compatibility problems. The term “peripherals” overlaps with what we typically refer to as display equipment in time-based media conservation. This is particularly so for software-based artworks with video outputs, which may make use of digital projectors, TV monitors and audio equipment commonly encountered in the care of video artworks (see Chapter 18).

Servers and Networking

While networking hardware overlaps with the types of hardware described in the preceding sections, it is discussed here in further detail due to its significance in networked and web-based artworks. A network is a connection between two or more computer systems through the use of shared communication protocols. The connection allows the connected systems to share and exchange resources, such as files and web pages. Networks can operate at a variety of scales, ranging from a local area network (LAN) between devices in physical proximity, to the Internet; the global network of interconnected networks which provides access to the World Wide Web. Networking capabilities may be provided by components built into computers (such as ethernet and Wi-Fi adapters) and extended through attached peripherals (for example, a network switch used to route LAN traffic between several connected computers).

Servers are computers which are used to serve content to other computers over networks. One computer — a client — connects to another — the server — over a network, allowing for the exchange of information between the two. Servers can be used to provide access to web content, route email, or provide access to databases and computational resources. Servers contain similar hardware components to desktop computers. While they may use a form factor which allows for them to be rack-mounted, any computer can be used as a server providing it has networking capabilities and can run a suitable operating system. Artists may operate their own servers or use servers provided by a third-party service provider. Where servers are employed, they can present a significant long-term preservation risk if not properly managed. Providing long-term public access to servers (which may be required for web-based artworks) requires that the server be maintained and updated, or otherwise risk loss of access and reduced security. The extent to which it is possible or desirable to move servers under the control of an institution is one important factor to consider.

Custom Hardware

Artists may use custom-built hardware to realize software-based artworks. Custom hardware is likely to be unique and can be contrasted with off-the-shelf hardware, which is industrially produced and sold at volume. Instead, bespoke circuit boards are created to meet a unique specification. These can be built by hand using a fixed base (such as breadboard or stripboard) and suitable electronic components. They can also be designed using electronics design software and sent to a third-party for manufacture and assembly. As custom hardware is likely to have been made to carry out a specific purpose within a software-based artwork, understanding it, and having the necessary documentation to rebuild it, if necessary, is very important.

Artist Jeffrey Shaw has frequently used custom hardware in his work, particularly bespoke human interface devices. For example, in *The Legible City*, 1989-1991, gallery visitors ride a bicycle mounted in front of a projection screen to navigate the virtual environment projected in front of them. This work relies on a custom-made analog-to-digital converter, the documentation and replication of which was a critical issue in a treatment of this undertaken by ZKM (Lintermann et al. 2013).

In addition to being built from scratch, off-the-shelf hardware can be modified by artists in order to alter the way it functions. Understanding the nature of the intervention and how it might be recreated is very important if artworks incorporating modified hardware are to be successfully preserved. Cory Arcangel is an American artist who has worked extensively with hardware modification, particularly his interventions in video game technology. This has included the modification of video game console controllers so that they self-play, in the case of *Various Self Playing Bowling Games* (2011), and a video game cartridge in *Super Mario Clouds* (2002) (Magnin 2015).

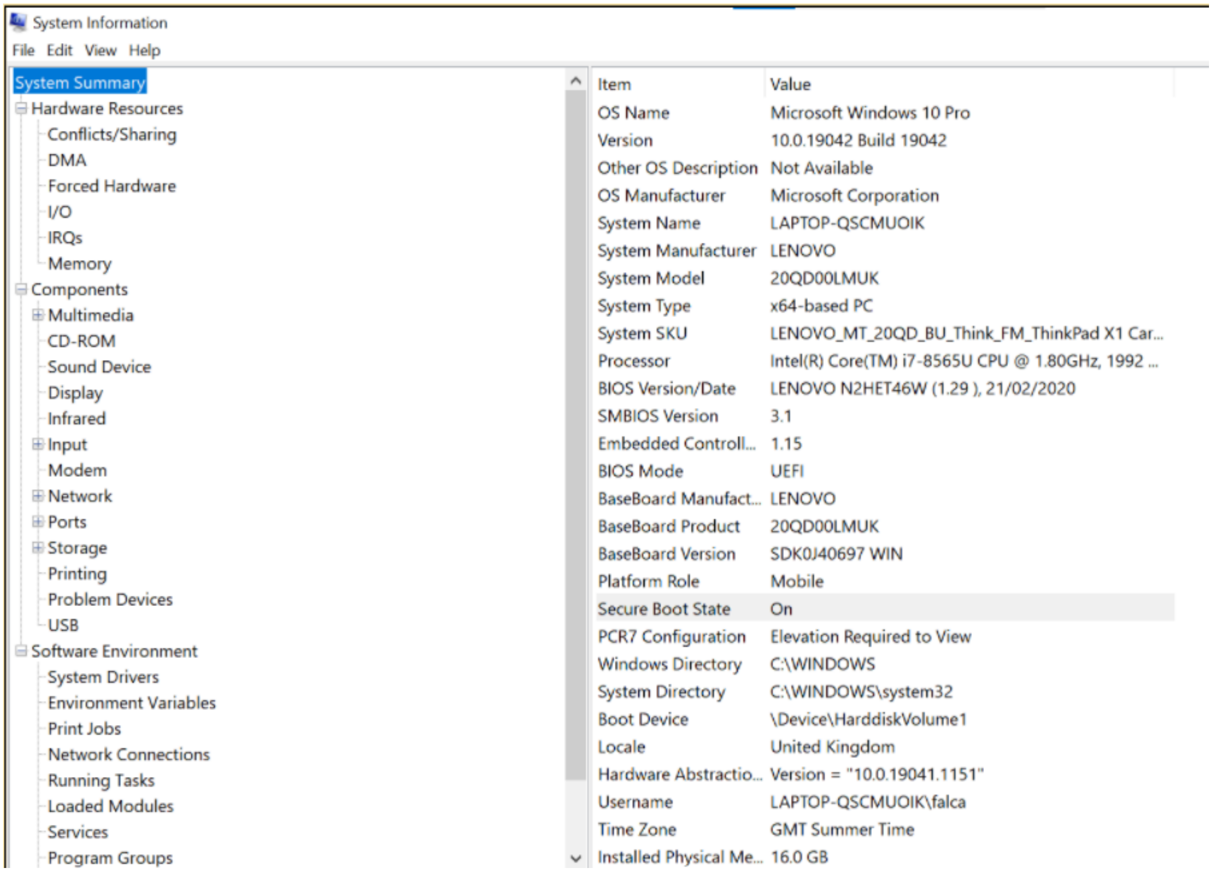
22.3.2 Analyzing and Documenting Hardware

Gathering information about hardware components should be done while the system is operational. Capturing the precise model and other relevant information (see Table 22.2) for all components will help to create duplicate back-ups, source future replacements, and define emulation configurations (see 22.4).

Gathering Computer Component Information

Off-the-shelf computers such as Apple's Mac Mini series or Intel's NUC series may have well-defined component sets associated with the specific model. In addition to capturing the manufacturers' specifications, the computer should be examined directly, e.g., to retrieve a list of components (including attached peripherals, operation system and installed software). Most contemporary operating systems include built-in tools, such as those packaged with Windows 10 and MacOS, both called System Information (see fig. 22.8). Third-party tools, such as HWiNFO for Windows (Malik n.d.), can provide more detailed information. Where possible, choose software tools capable of exporting the information they gather in a widely understood, non-proprietary format such as plain-text, XML or HTML. Storing analysis outputs in these formats increases the chances that this information remains accessible in the long-term. For microcontrollers and single-board computers, the relevant datasheet from the manufacturer should be located and retained.

<<INSERT FIGURE 22.8 HERE [
TBM_Chapter_22_Software_Figure_8_SystemReport.png] Caption:
Figure 22.8 The report produced by the built-in System Information tool on a Windows 10 laptop provides a detailed account of all hardware and software components. Screenshot: Patrícia Falcão



Item	Value
OS Name	Microsoft Windows 10 Pro
Version	10.0.19042 Build 19042
Other OS Description	Not Available
OS Manufacturer	Microsoft Corporation
System Name	LAPTOP-QSCMUOIK
System Manufacturer	LENOVO
System Model	20QD00LMUK
System Type	x64-based PC
System SKU	LENOVO_MT_20QD_BU_Think_FM_ThinkPad X1 Car...
Processor	Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz, 1992 ...
BIOS Version/Date	LENOVO N2HET46W (1.29), 21/02/2020
SMBIOS Version	3.1
Embedded Controll...	1.15
BIOS Mode	UEFI
BaseBoard Manufact...	LENOVO
BaseBoard Product	20QD00LMUK
BaseBoard Version	SDK0J40697 WIN
Platform Role	Mobile
Secure Boot State	On
PCR7 Configuration	Elevation Required to View
Windows Directory	C:\WINDOWS
System Directory	C:\WINDOWS\system32
Boot Device	\Device\HarddiskVolume1
Locale	United Kingdom
Hardware Abstractio...	Version = "10.0.19041.1151"
Username	LAPTOP-QSCMUOIK\falca
Time Zone	GMT Summer Time
Installed Physical Me...	16.0 GB

If software tools cannot be used, e.g., in older computers where analysis tools may not be available, a visual inspection of the physical components inside the computer can produce model numbers and/or serial numbers, which can then be cross-referenced with other resources to gather more detailed information. The interior inspection of the computer is also an opportunity to assess its condition and identify problems such as dust build-up or leaking capacitors. The contents of some computers may be difficult to access due to the way they have been assembled or the absence of screws. For example, the internal components of some Apple iMacs can only be accessed by prising off the glued-on LCD panel. Careful consideration should be given to balancing the possibility of damaging the computer against the value gained through this intervention. When handling hardware components with exposed circuitry, there is a risk of damaging components through electrostatic discharge (ESD). To avoid this, use a combination of ESD mitigating equipment such as anti-static mats, wristbands, and gloves (see Chapter 14 for further guidance).

Documenting Peripherals and Custom Hardware

Peripheral devices, such as cameras, sensors, human interface devices (e.g., controllers) and other off-the-shelf hardware, should be recorded — including noting and photographing the interface used to connect to the computer. Any manuals or other relevant documentation available from the manufacturer should also be collected where possible. Peripheral hardware may be accessed using specific drivers, so it is important that this software be identified, documented, and appropriately archived.

Custom-made hardware devices require particularly detailed documentation. As a general rule, aim to gather enough information that they could theoretically be reconstructed from scratch. Technical documentation to seek includes schematics, circuit diagrams and bills of materials. If this documentation is not available, as much of it should be generated as possible at the point of acquisition, ideally in collaboration with the artist and inspecting the circuits as necessary. It is also important to identify whether hardware has been programmed with custom code that cannot be easily retrieved (e.g., a microcontroller), as this should be acquired separately. If not available, measures can be taken to retrieve it using interfaces attached to the microcontroller or to reverse engineer its function through use of a logic analyzer (Fino-Radin 2016; Bunz 2018).

Human interface devices may require more extensive documentation of user experience. Factors to consider include the array of possible interactions, use of haptics, look and feel of materials used in the interface's construction, and the relationship between the movement of the interface and its effects on the software. Documenting interface devices can be achieved by creating videos of user interaction. Care should be taken to ensure as many aspects of interaction are captured as possible, and whether video capture would benefit from comments, narration or further written explanation. The materials that constitute the device should also be identified and appropriate steps taken to care for them; this is work which can be undertaken with a specialist conservator with suitable knowledge of the materials.

22.3.3 Hardware Failure and Obsolescence

Failure of hardware components occurs due to damage to or degradation of their electronic circuits or mechanical components. This can be caused by faulty or degraded components, environmental conditions (such as extreme heat or damp), power surges and electro-static discharge (ESD). Based on discussion with a number of conservators with experience maintaining hardware — Candice Cranmer and Nick Robinson (ACMI, Melbourne), Chris King (Tate, London), Paul Jansen Klomp (media artist / tutor Media Art at Artez, Enschede), Arnaud Obermann (Staatsgallerie, Stuttgart), Morgane Stricot and Matthieu Vlaminck (Zentrum für Kunst und Medien, Karlsruhe) — the following were identified as common sources of failure (Cranmer and Robinson 2021; King 2021; Klomp 2021; Obermann 2021; Stricot and Vlaminck 2021):

- **Battery failure**. For example, reduced ability to hold charge or leaking battery acid. This is particularly challenging to address where these are soldered to the motherboard and therefore difficult to replace. Leaking battery acid can also damage the circuit board and adjacent components.
- **Capacitor failure**. For example, reduced ability to hold charge or leaking electrolytic fluid. The latter is particularly common among batches of capacitors from the early 2000s, where a build-up of gas in a faulty electrolytic capacitor can cause the shielding to split.
- **Failure of a mechanical component** such as a fan, switch, or drive. For example, the motor that spins the magnetic platter inside a magnetic hard disk drive may fail, rendering the data contained inaccessible.

- **High operating temperatures.** Factors such as high ambient temperature and failure of cooling systems (e.g., accumulation of dust, fan failure) can result in operating temperatures exceeding component tolerances, which can accelerate degradation of materials.
- **Oxidation of connectors** between components, which impedes connectivity. For example, this can occur in the metal contacts that connect an expansion card to a motherboard.
- **Poor storage or display conditions**. For example, exposure to humidity and accumulation of dust and other debris can cause components to short-circuit when powered on.

There is little publicly available empirical data on the failure rates of computer hardware, making it hard to reach meaningful conclusions as to its frequency in well-controlled conditions. Given the long time spans over which those collecting software-based art aim to preserve it, it may be assumed that hardware failure will eventually occur in any given component due to the tendency of all objects to deteriorate over time. In the examples of sources of hardware failure listed above, these problems can often be addressed, although in all cases specialist equipment and expertise would be required. A faulty capacitor can be identified through visual inspection, and then replaced before it causes damage. A magnetic disk platter can be removed from a failed drive in controlled conditions, and the data recovered. Other types of hardware failure may be more catastrophic making repair impossible, particularly due to the industrial processes by which circuits are manufactured. The extent to which failures are addressable can be identified by balancing the benefit against the resources required.

When faced with hardware failure, it may be more cost- and time-effective to replace a component rather than repair it. However, this becomes increasingly challenging with hardware obsolescence, whereby hardware components are no longer sold or supported by their manufacturer. Hardware obsolescence is driven by commercial interests. Newer products may provide better performance or additional features which make them more appealing to the consumer, thus shifting demand from an old product to a newer product and disincentivizing the manufacturer from continuing production. In other cases, a manufacturer may simply choose to stop producing or supporting a product, despite continued consumer interest. The effect is that hardware becomes both increasingly difficult to repair, as expertise dwindles with decreased financial incentive, and to replace like-for-like, as the now fixed number of components become increasingly rare. This forces replacements to be made using different hardware: the treatment approach is called *hardware migration*. This process has a secondary effect on the viability of software due to the changing hardware support in operating systems: vendors are unlikely to continue to provide driver software for hardware that is no longer commercially available (see 22.3.7 for software obsolescence, and 22.4.3 for hardware migration).

22.3.4 What is Software?

Software consists of stored instructions — or *code* — written in programming languages. These are formal languages used to tell a computer what to do. Software can run on many different hardware platforms, from small devices such as micro-controllers to mobile devices, tablets, desktop computers and servers. In some cases, the software is written to run on devices of any size and in other cases, the software is too closely aligned with a specific hardware device or custom hardware to run on anything else. Web-based software runs on a webserver using the programming languages, media file types and other paradigms appropriate for a web-server environment.

Programming Languages

Programming languages have evolved over time reflecting the evolution of the hardware, the role of computers and how they are used in society, and to meet specific computational and usage requirements throughout all areas of study and professional practice. There are several hundred programming languages (Rösler 2021); those most frequently encountered in software-based art are introduced in 22.3.8.

Source code is a human-readable set of instructions written in a specific programming language. It can be read and/or edited in any text editor and is commonly written using a software program called an IDE (Integrated Development Environment) which is designed to support the software development process. Source code can also be developed within some *WYSIWYG* environments such as Adobe's Flash.

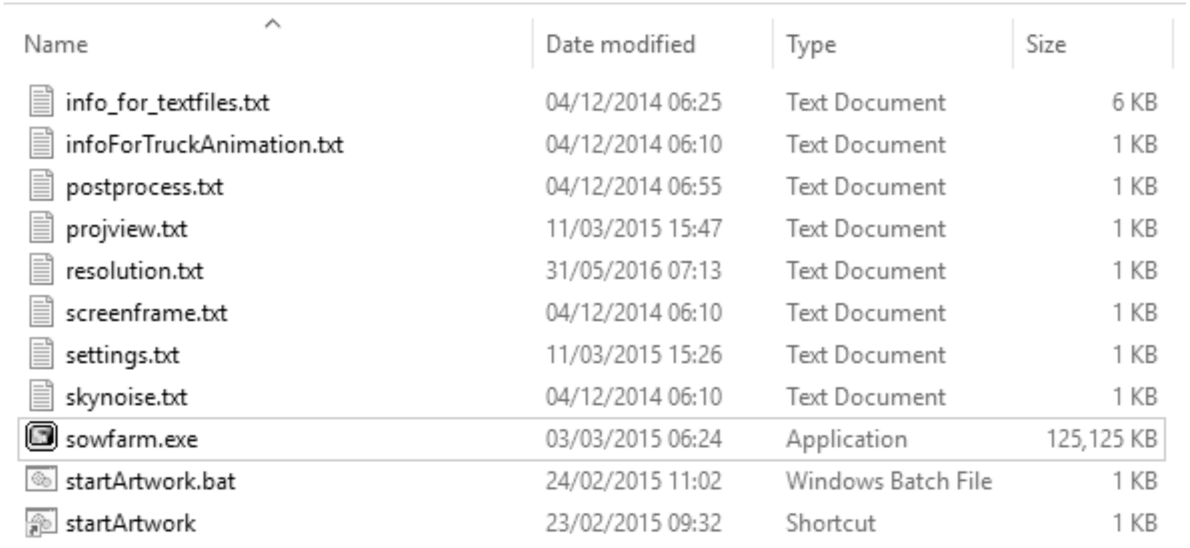
The term *high-level programming language* refers to programming languages that remain distant from the details of the computer in order to make programming easier so that onerous tasks such as memory management are handled "behind the scenes". High-level languages are meant to be human-readable; using a high-level programming language, artist-programmers can focus on the features of code to write the instructions for their artwork using variables, Boolean logic ("if - then" structures), iteration (such as "loops"), functions and other features of the language that are not specifically dictated by the computer's architecture.

In order for a computer program to be executed, it must be transformed into instructions — called *machine code* — which can be understood by the target computer. Machine code is a low-level programming language with close correspondence to the actions carried out by the computer hardware. The point at which this transformation happens varies and is usually dictated by conventions associated with the programming language used. While for interpreted languages like PHP the source code is sufficient to execute the program, for languages such as C++ and Java, the source code is compiled to executable code, which is not human readable. In cases where the source code gets compiled, it is no longer needed to run the software.

Executable Software

Executable software is the collection of the executable code and associated data required to run a program. This is stored in and managed as files, ranging in complexity from a single executable file to a collection of files which are called upon when the software is run (see fig. 22.9). This file or collection of files can be referred to by a number of other similar or related terms including *application*, *binary* and *build*. The contents of executable files are stored in forms optimized for computer processing and are not intended for human reading.

<<INSERT FIGURE 22.9 HERE [**TBM_Chapter_22_Software_Figure_9_SowFarmAppDirectory.png]** Caption: **Figure 22.9** Application directory for John Gerrard's *Sow Farm (near Libbey, Oklahoma)* (2009), viewed in Windows 10, containing a Windows Portable Executable (.exe) file, settings stored in plain text files and batch and shortcut files for running the software. Screenshot: Tom Ensom



Name	Date modified	Type	Size
info_for_textfiles.txt	04/12/2014 06:25	Text Document	6 KB
infoForTruckAnimation.txt	04/12/2014 06:10	Text Document	1 KB
postprocess.txt	04/12/2014 06:55	Text Document	1 KB
projview.txt	11/03/2015 15:47	Text Document	1 KB
resolution.txt	31/05/2016 07:13	Text Document	1 KB
screenframe.txt	04/12/2014 06:10	Text Document	1 KB
settings.txt	11/03/2015 15:26	Text Document	1 KB
skynoise.txt	04/12/2014 06:10	Text Document	1 KB
sowfarm.exe	03/03/2015 06:24	Application	125,125 KB
startArtwork.bat	24/02/2015 11:02	Windows Batch File	1 KB
startArtwork	23/02/2015 09:32	Shortcut	1 KB

Executable code is stored and managed in a variety of formats, which may be accompanied by a wide variety of other files. A useful starting point for understanding these collections of files is to examine the folder-file structures (including file and directory names which may be indicative of meaning) and the file formats present. Somewhere within the collection of files will be one or more executable files, which are used to run the software on a suitable host computer.

Successfully running the executable software relies on connections between this executable software and other software and hardware components, which were embedded in the executable software during its development. These are known as dependency relationships. The importance of dependency relationships in the realization of the software-based artwork may vary. In some cases, access to a highly specific component might be critical and its absence or failure may cause the software to fail to execute altogether; for example, a driver required by the software to communicate with a hardware device. In other cases, there may be some flexibility in the component used; for example, software may require access to a graphics processing unit (GPU) to render 3D graphics but may not need access to a specific model. Dependency relationships can also extend to external resources, such as web services and APIs. See section 22.3.6 for further guidance on identifying dependency relationships between components.

22.3.5 Types of Software

Custom Software vs. Off-the-shelf Software

For contemporary artists who create software-based works of art, custom software is their primary medium. Artists may have their own characteristic style with respect to selecting programming language(s); developing and coding the algorithms that drive the appearance and behaviors of the artwork; and stylistic programming choices, e.g.whether to calculate color shifts using an HSB color model rather than RGB model or whether to include comments and thereby retain earlier “drafts” of source code within the code. In other cases, artists may work with a collaborator, or team of collaborators, who will also influence the technical choices made and the style of programming.

Off-the-shelf components are software components which have not been made specifically for the software-based artwork. Instead, they are typically pre-packaged, widely distributed software components designed to meet some common need. For a software-based artwork, off-the-shelf components usually consist of at least an operating system — itself incorporating many individual software programs — but may also include additional libraries, Application Programming Interfaces (APIs) and drivers which are not part of the operating system by default. These components serve critical supporting roles in the execution of the artist-created software and form a layer that connects the artist-created software with the hardware environment.

Web-based Software

Websites run software that is configured to run on a web server (see 22.3.1) and accessed by browsers. Websites often require the use of more than one technology (e.g., a programming language such as JavaScript and the use of a markup language such as HTML) and they can also host any required media files such as digital images, audio and/or video files.

Proprietary vs. Open-source Software

Proprietary software is under copyright by a publisher and available only to licensees. In this case the vendor might set limitations such as the number of installations permitted for a given license and force that limitation through product keys, a serial number, online authentication or other means.

Free and open-source software is provided at no cost with access to the source code. It can be released under a variety of licenses: license such as the public domain Creative Commons CC0 license (Creative Commons (CC) n.d.); a permissive license such as the MIT license (Open Source Initiative n.d.); or under Copyleft such as the GNU license (GNU Operating System n.d.). Use of open-source software benefits preservation, as source code can easily be gathered and reused, thus ensuring that it can be modified for preservation purposes if necessary (see 22.4.8).

22.3.6 Analyzing and Documenting Software

Source Code Analysis

Artwork inspection can provide a good understanding of the make-up of a work, but not all behaviors are discernable to human observation by simply running the piece. Complex features like open durations, randomization or generative functions can only be identified through source code analysis. A close reading of the source code, which contains the instructions and the algorithms that create the artwork not only reveals the artist's decisions on the behaviors and presentation of the work, but also its technological make-up and genesis.

This examination method requires the expertise of a programmer or computer scientist and has been introduced to the conservation community in recent years through interdisciplinary efforts between media conservators and computer scientists (Engel and Wharton 2014, Engel and Wharton 2015, Guggenheim Blog 2016, Engel and Phillips 2018, Engel and Phillips 2019).

When conducting source code analysis, the examiners try to find answers to the following questions:

- Color: How does the artist stipulate the use of color (e.g., HSB or RGB)? How are color values calculated?
- Randomization: Does randomization play a role in the work? What is the impact on the artwork's behavior and appearance based on the random results?
- Speed of animation: Is animation speed rendered or measured in absolute terms such as seconds or milliseconds which are hardware-independent or relative terms such as delays that reflect processing time?
- Input: what is the role of user input, if any? Does other input such as the ambient sounds in the exhibition space play a role?
- Sound and image creation: Are sounds and images created dynamically as the software is running or does the artwork use external media files that are played or displayed?
- Data sources: Does the artwork use one or more data sources and are they static or dynamic? How are the data retrieved and processed? Are the data conserved for study or future exhibitions?
- Systems issues: The interaction and dependencies between the artwork and the operating system, peripherals and the role of configuration are defined.
- Web--based artworks: What are the parameters for responsive design (the specific page layouts for a mobile device, tablet, or desktop)? What are procedures for permissions and access that requires authentication? Does the software address specific browser constraints?

The following conservation concerns and preservation risks can be identified in the source code:

- Programming languages and third-party libraries: Use source code analysis to identify all of the programming languages and all of the external, third-party libraries that should be conserved along with their source code.
- Technologies: Use source code analysis to identify all of the technologies used in a given work including device drivers for hardware and hardware requirements noted.

- Cost and resource analysis for preservation and interventions: Use source code analysis to outline the scope and skills required to plan time and cost budgets for interventions.

Source code analysis - that is, reading and documenting the source code of an artwork - can provide a rich resource not just for conservation, but also for art historical, technical art historical and curatorial study. For conservators, source code analysis and documentation can offer an understanding of the behaviors of an artwork as well as insights into potential preservation risks an artwork may be facing. It helps to assess how much work is required to make a piece run again; which technical skills are required to do so; and the scope of such a project in order for the conservator to make informed decisions on intervention and conservation treatment.

Software documentation is essential to long-term maintenance and thus has a long history and is a major activity in software engineering so there is a great deal of literature on this practice (Garousi et al. 2013; Bavota et al. 2019). Source code documentation can take a number of different formats, depending on the resources available with respect to time, cost, and the expertise of those writing the documentation and the goals of the project e.g., whether the project is meant to benefit conservators, curators, and art historians, and/or future programmers who may be commissioned to work on the piece. Successful approaches include making a study copy of the source code and annotating the study copy with comments and explanations for future technologists; writing a descriptive narrative of each logical section of the source code for curatorial and art historical readers; and building out tables and diagrams that identify specific aspects of the code. Tables or spreadsheets can be used to list a color vocabulary, the media file types used in the artwork, the static image files used and other facets of the artwork to complement documentation that identifies the specific areas of the source code that constitute risk such as deprecated code (Phillips et al. 2017). Visualizations such as flowcharts are appropriate for older works such as those written in Pascal, while UML diagrams are commonly used with object-oriented programming languages such as Java (Engel and Wharton 2014; Engel and Wharton 2015).

On the value of source code analysis:

Glenn Wharton: Source code analysis is a new tool in the conservator's technical research kit, but it is not yet widely used. It is a standard maintenance procedure in software engineering, but as with many research methods adapted from other industries, it needs to be applied with a strong understanding of conservation ethics. Now that a new generation of conservators are being trained to work on time-based media, no doubt some will arrive in museum labs with programming skills. Yet the field has always relied on technical expertise from other fields, and I predict that software engineers will increasingly be consulted in software-based art conservation. (Wharton 2021)

Christiane Paul: The degree to which artists annotate their source code varies and source code analysis helps to determine the purpose, functionality, and intent of the code. Source code analysis has a tremendous impact in that it helps to identify the conservation strategies—from rewriting code for a migration to emulating the environment—that best maintain the original artwork's functionality and intent. The Whitney Museum still is in the early stages of conducting source code analysis of the works in its artport collection, the Whitney Museum's platform for net art, and, so far, this analysis has been undertaken only in collaboration with classes at New York University where students have performed the work. The results have not yet been actively used but already were very helpful in determining potential strategies for conserving software-based work. (Paul 2021)

Claudia Roeck: Source code analysis helps to understand and value the materiality of the artwork. It has the same purpose as scientific imaging and chemical analysis of a painting. Source code analysis contributes to a better understanding of the artist's and/or programmer's intent. Together with interviews of the artist and/or the programmer one can better decide which algorithms or features are significant and have to be maintained. Source code analysis makes one more aware of the craftsmanship of the programmer/artist and of the community culture linked to certain digital tools. Even if a work has to be transferred to a new technology, the knowledge of the algorithms, program features, and parameters used in the source code informs the choice of programming language and platform and facilitates reprogramming. When acquiring software-based artworks, we do source code analysis to estimate whether and how long the work will be functional and what additional information or tools we need to preserve it. Source code analysis in the case of *TraceNoizer* (2001) by LAN (Annina Rüst, Fabian Thommen, Roman Abt, Silvan Zurbruegg and Marc Lee) (Roeck n.d.) supported our decision-making regarding preservation strategies on a technical level and also on a contextual level (what kind of technology was used in 2001 and how the technological landscape has changed since, what that means for the interpretation of the artwork and so forth). (Roeck 2021)

Glenn Wharton
Professor, Department of Art History & Lore and Gerard Cunard Chair, UCLA/Getty Program in the Conservation of Cultural Heritage, University of California, Los Angeles

Christiane Paul
Christiane Paul, Adjunct Curator of Digital Art, Whitney Museum of American Art and Professor, School of Media Studies, The New School, New York

Claudia Roeck
Claudia Roeck, PhD candidate, University of Amsterdam, Media Studies and time-based media conservator

Executable Analysis

Executable software typically contains compiled code, which is not human-readable and makes analysis and documentation work more difficult if the source code is not available. Nonetheless, in some situations executable code is the only representation of the artist-created software available. It is important to note that this may significantly increase the risk of losing access to this artwork in the future, as it makes applying some preservation strategies difficult, if not impossible. In this section we will briefly discuss the options available in such situations and the limitations to what we can learn about a computer program and the way it functions and behaves from executable code alone.

Unlike other common file formats in time-based media conservation like digital video, executable files usually contain little in the way of accessible metadata. Instead, specialized tools such as debuggers and decompilers are required to extract useful information from them and even when successful may offer limited insight compared to full source materials. The use of such tools leads us down the path of software reverse engineering — that is, attempting to gain information about how the software works with limited prior knowledge about how it was created. At its most complete, reverse engineering would aim to produce source code similar or equivalent to the original source code, from the executable code. In practice however, this is a laborious and highly specialized activity, particularly for larger and more complex programs. One approach that may come close is decompilation: the use of a software tool called a decompiler to create an approximation of the original source code from the executable software. The success of decompilation will depend on the characteristics of the software (e.g., a program compiled to machine code is harder to decompile than a Java executable) and availability of a decompiler tool. Approaches to reverse engineering software-based artworks are discussed further in Ensom (2018).

Documenting a Software Environment

When documenting a software environment, identify which operating system is being used and any additional off-the-shelf software such as device drivers, runtime libraries and database software. Documentation should include the specific version and build number of each program where possible. Most contemporary operating systems include built-in tools for reporting this, such as the tools (both called System Information) packaged with Windows 10 and MacOS (see fig. 22.8 and 22.3.2). Where possible you should choose software tools capable of exporting information in a widely understood, non-proprietary format such as plain-text, XML or HTML. Alternatively, it may be possible to gain equivalent information from examining contents of storage media (e.g., a disk image). This method is less reliable, as it requires the conservator to manually locate installed programs on storage media. While there are typical install locations (e.g., Windows 10 uses the Program Files directory, while MacOS X uses the Applications directory), this is something which can easily be customized by the artist or their collaborators.

It is important to note that while the software environment is distinct from the artist-created software, the default configuration of off-the-shelf software components can be further modified by the artist. For example, web server or database software requires a level of configuration to be used that can then impact its connections with other software components. In the case of John Gerrard's *Sow Farm*, the artist applies image processing techniques using the graphics card driver, which significantly impact characteristics of the projected image (Ensom 2019). The extent of any custom configuration of off-the-shelf components should therefore be identified and documented. Given the large number of configuration options available for some software, this can be difficult to identify through examination alone, so it is best to work closely with the creator of the system — be that the artist or a collaborator — to identify these and their significance.

22.3.7 Software Obsolescence

As hardware and operating systems evolve over time, the risk of software obsolescence grows. External libraries are updated or no longer supported; devices become outdated or drivers are no longer written for current hardware and operating systems; programming languages develop and change and may not be downwardly-compatible; and other changes in the system will cause software applications to run in ways that are not anticipated, to exit prematurely as a “crash” or not to run at all. This can be managed in the short-term by avoiding making changes to the software environment (e.g., preventing automatic updates, creating disk images). However, when hardware fails or an external dependency changes, obsolescence of components will have to be negotiated.

Web-based artworks are a clear example of works subject to software obsolescence and software incompatibilities that can arise when the underlying operating system and/or the software environment are updated or changed in ways that prevent the artwork from running as expected. Updates to operating systems, programming languages and other widely used software resources are often publicly announced ahead of time and in a best-case scenario, risk assessment and interventions of the affected artworks can be considered while the artworks still run. Web-based artworks are also subject to updates in browser software and should be tested periodically using the most popular browsers available at that time to ensure compatibility.

22.3.8 Programming Languages and Platforms and their Preservation Implications

Many artworks employ more than one programming language in order to best handle the tasks that the artist wishes to accomplish. It is important to understand which language(s) are in use and the specific role(s) that they play in order to plan for long-term conservation and re-exhibition. In the following section, several categories of programming languages are introduced along with examples of languages that conservators may encounter when working with contemporary artworks. The details listed here are not meant to be a comprehensive software review or to replace the extensive manuals and documentation on these languages available in print and online, but rather to point out specific features, functionality, behaviors, and risks associated with each of these languages or language groups in the context of art conservation.

High-level Programming Languages that are Compiled

This group of programming languages includes languages such as Java, C, and C++. Compiled programs are architecture-specific, meaning that they run on a computer that has an architecture (MacOS, Windows, etc.) corresponding to the machine code language used. Risk assessment of these works includes taking care to find out not only which programming language was used but also which specific version of that language and whether any additional libraries of source code were included when the software was compiled.

High-risk artworks in this category include the many Java Applets written for web-based applications in the 1990s and early 2000s as Java Applets are no longer supported by most browsers. Works in C or C++ are at risk due to current or future hardware obsolescence as the close pairing of the language with the operating system and hardware will render these works inoperable when the appropriate hardware is no longer functioning or available.

High-level Programming Languages that are Interpreted; Scripting Languages

There are a number of programming languages that do not require compilation but rather require an interpreter. An interpreter executes the human-readable source code at the point at which the program is run. This requires an installation of a suitable interpreter to be available to the software, such as Python in order to run a Python script. Client-side JavaScript is an example of a scripted programming language that can run locally or on a web server through a browser; either way, viewing source within a browser will typically reveal the JavaScript source code directly to a viewer in a human-readable format.

Server-side web-based scripting languages such PHP, Perl or Python are interpreted but the source code is not accessible to the institution or collector unless the artist provides access to the server or the files. Further, scripts that handle tasks such as file management are hidden from view and must also be specifically provided to the collector or institution.

Preservation measures for artworks in this category include collaboration with the IT staff managing the web server to maintain a stable and secure environment, such as consistency in the appropriate path designations to ensure that the artworks run in the correct programming language and version. Software upgrades can cause incompatibilities and failure; for example, while PHP is typically downwardly compatible, Python 3.x is not, and software written in Python 2.x will not run or will not run correctly in Python 3.x.

High-risk artworks in this category include artworks that rely on widely used libraries such as server- and client-side JavaScript libraries that change over time. In these cases, a virtual development space in which a conservator or their team can test the artwork in a software environment that is configured to match the updated languages and libraries of the production server is crucial to ensure the ongoing successful presentation of these works.

Markup Languages

Markup languages such as HTML (*Hypertext Markup Language*) are not programming languages but rather conventions of annotation that allow implementation to differentiate content from format; to differentiate content from document structure; and in some cases, to identify content roles such as designated text as a citation using the <cite> … </cite> tags in HTML, also known as *semantic markup*. HTML files are human-readable and may be associated with CSS (*Cascading Style Sheet*) files which describe the rules as to how content is displayed with respect to page layout, font size and color and other format specifications, whether in print, on a tablet, mobile device, or other device. As text files, HTML and CSS files are easily stored and preserved. These languages change slowly over time, often with lengthy announced lead-times which gives conservators time to address any necessary software updates such as replacing tags that may be deprecated. For example, a list of tags noting which ones are deprecated is available on the w3schools website (W3Schools - HTML Element Reference n.d.); additional documentation is available on other sites as well.

Risk assessment of these works includes documenting the versions of the mark-up languages used and identifying deprecated tags and a remediation strategy where necessary. Risk assessment will also include checking for cross-browser compatibility to ensure that a given work either opens in multiple browsers or that a user is advised *which* browser(s) to use when viewing a work in this category if limitations are found.

Query Languages

Query languages are used to manage data and answer questions posed to a specific database. Many artworks use data in databases, which are retrieved by running queries against either a dynamic or a static database. The queries themselves may typically be launched via a program written in a programming language such as Python, PHP, Perl, or another language.

Preservation measures begin by identifying the type of database and its storage location along with verifying access and permissions to ensure security.

SQL (Structured Query Language) is a query language used to obtain data and results from a server-side relational database such as MySQL, Oracle, PostgreSQL or from SQLite with its smaller footprint running either locally or on a server. Standard database documentation which is useful to the conservator includes a data dictionary (a list of all of the tables, fields, and other objects in the database with descriptions); an E-R diagram (an "entity-relationship" diagram which provides a visualization of the relationships among the tables); and the results of a standard "DUMP" query. A "dump" query such as "mysqldump" in MySQL records the contents of the database as a human-readable script file for study and analysis or to create a new copy of the database.

SPARQL (SPARQL Protocol and RDF Query Language) is used to retrieve and manipulate data in a Resource Description Framework (RDF) format such as in Wikibase implementations or with Wikidata.

A decision must be made as to whether an artwork will run against a static (i.e., unchanging) database e.g., the data available when it was first exhibited; or a dynamic database which is updated each time the work is run. If the artwork will run against a static database in the future and is "frozen in time", then the data should be backed up and conserved for future use. If a decision is made to run the artwork against a dynamic (changing) database, then the software that is used to obtain and manage the database becomes an integral part of the conservation effort and preservation measures going forward should include identifying all of the data sources and their respective availability. For example, if external data are captured from one or more websites, the data sources and/or data formats and availability will likely change over time, creating risk to the work's long-term preservation. In such cases it is advisable to store copies of the database at appropriate intervals so that the work can be exhibited in the future with historical datasets if necessary, and if so, agreed by all of the appropriate stakeholders.

Assessing and displaying artworks that store data from user input should ideally be coordinated with the IT professionals involved to evaluate security and to protect against provocative, illegal, or threatening entries by users. In many countries, it is important to consider General Data Protection Regulations (GDPR), if any identifiable personal data (e.g., name, age, contact details) is or can be supplied by users. For artworks that collect statistics and data based on user activity; it may be important to retain these datasets for future study.

Multimedia Application Authoring Languages and Platforms

There are some programming languages which run inside of software applications and have been used by artists. *Lingo* is a scripting language developed for use with Macromedia's and later Adobe's *Director* platform in the 1990s. Director files (.DCR) could be output as executables to run directly on Windows or MacOS or as content for the *Shockwave* runtime environment on the web at that time. *ActionScript* was developed for use with Adobe's *Flash* to run using Adobe's *Shockwave* player on the web. In both cases, conservators would need the appropriate version of either Director for Lingo or Flash for ActionScript to set up a development environment that will allow study, further development, or re-compiling if necessary. ActionScript 3.0 is not backwards compatible so special care must be taken to correctly identify which version of the language is in use in order to set up the development environment. Artworks that use these formats are at risk.

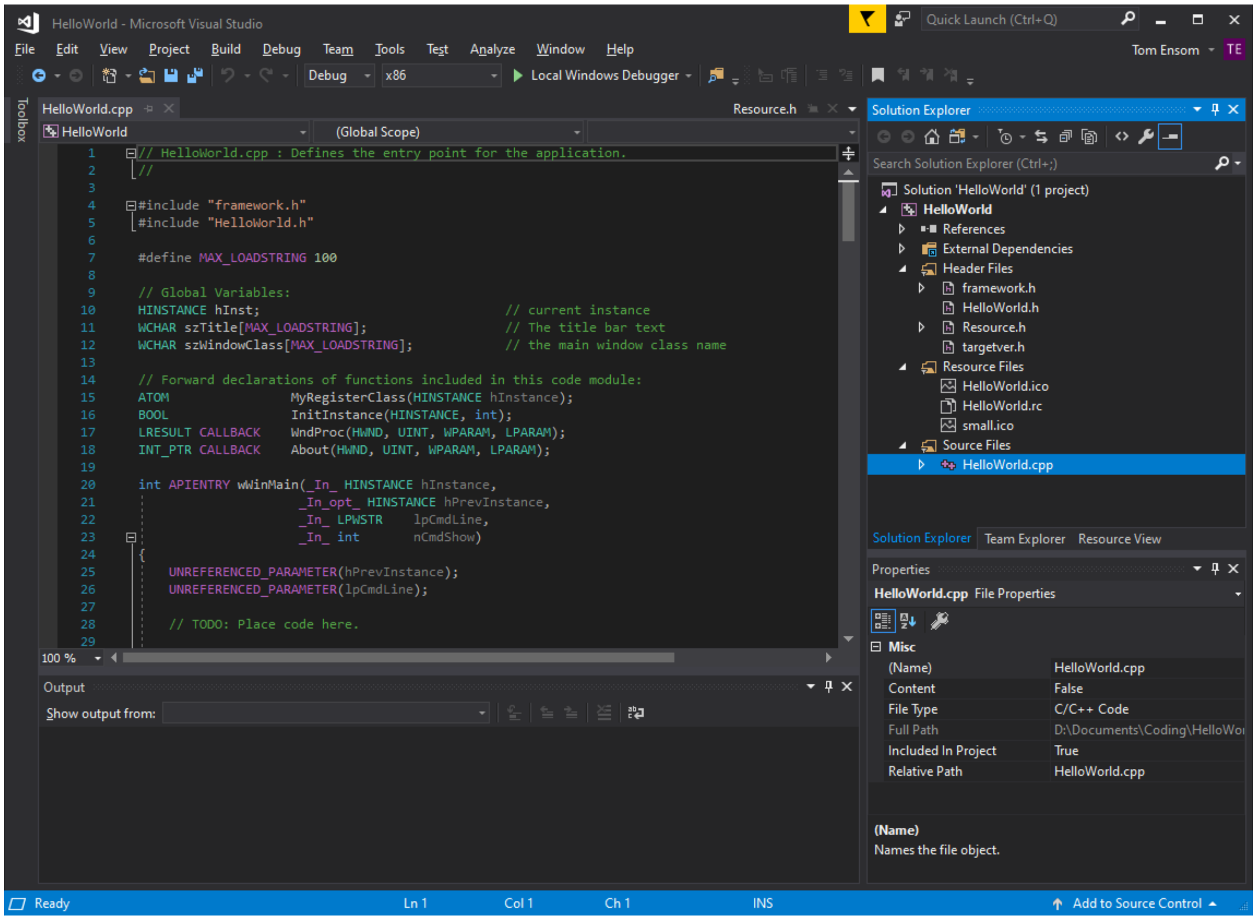
Max (also known as Max/MSP/Jitter) by Cycling 74 and Pure Data (Puckette and Various 1996-2021) are used to build multimedia applications. Both use a visual programming language in which the programmer generates code by manipulating a graphical environment rather than by writing code textually. Unity and Unreal Engine are game development platforms that primarily use the programming language C#.

Risk assessment of artworks built with these languages includes an evaluation of whether the work runs on a local mac or PC as a compiled version and if that is sufficient for re-exhibition (Engel and Phillips 2019; see Section 22.4).

IDE's (Integrated Development Environments)

An IDE is an *integrated development environment;* a category of software that is used to support software development. They may be language specific, such as NetBeans for Java and PyCharm for Python, target a specific platform such as Microsoft Visual Studio for Windows development, Xcode for Mac development, or the Arduino IDE for Arduino microcontrollers. They can also target a specific group of users such as Processing, an IDE created for artists.

<<INSERT FIGURE 22.10 HERE [**TBM_Chapter_22_Software_Figure_10_VisualStudioIDE.png]** Caption:
Figure 22.10. A simple C++ application project opened in the Microsoft Visual Studio 2017 IDE. Screenshot: Tom Ensom



```
// HelloWorld.cpp : Defines the entry point for the application.
//

#include "framework.h"
#include "HelloWorld.h"

#define MAX_LOADSTRING 100

// Global Variables:
HINSTANCE hInst;                                // current instance
WCHAR szTitle[MAX_LOADSTRING];                  // The title bar text
WCHAR szWindowClass[MAX_LOADSTRING];            // the main window class name

// Forward declarations of functions included in this code module:
ATOM                MyRegisterClass(HINSTANCE hInstance);
BOOL                InitInstance(HINSTANCE, int);
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK    About(HWND, UINT, WPARAM, LPARAM);

int APIENTRY wWinMain(_In_ HINSTANCE hInstance,
                     _In_opt_ HINSTANCE hPrevInstance,
                     _In_ LPWSTR    lpCmdLine,
                     _In_ int       nCmdShow)
{
    UNREFERENCED_PARAMETER(hPrevInstance);
    UNREFERENCED_PARAMETER(lpCmdLine);

    // TODO: Place code here.
```

Misc	
(Name)	HelloWorld.cpp
Content	False
File Type	C/C++ Code
Full Path	D:\Documents\Coding\HelloWo
Included In Project	True
Relative Path	HelloWorld.cpp

(Name)
Names the file object.

Preservation measures for these works include documenting which version of the IDE was used and with which programming language. Using Processing as an example, earlier versions of Processing do not support all of the programming languages currently available, and some versions are not backwards compatible, so it is important to collect the appropriate version of Processing for each artwork for further documentation and treatment.

22.3.9 Specific Artwork Genres and Preservation Considerations

Web-based Art

Web-based artworks range in complexity from the static HTML web pages (see 22.3.8) of the early web, to those making use of technologies like JavaScript and Flash to create interactivity, to large sites that are dynamically generated by software running on a web server.

A web-based artwork may include one or a combination of high-level scripting languages and/or markup languages with the possible addition of a query language if a database is used for the artwork in addition to any media files. Some of the web technologies used by artists in the past are at particular risk at this time. This includes Java Applets and Adobe Flash content, which no longer runs in contemporary browsers, and the loss of functionality associated with the removal of support for specific HTML tags.   Other considerations include loss of functionality for browser-based media playback, which changes over time as support for old formats is removed. Browsers are updated and modified regularly, and new browsers are introduced, which may not correctly display earlier web-based artworks.

Preservation of a web-based artwork can be approached from the perspective of files stored on the web server (server-side) or the display-ready web pages returned by the server (client-side). When implementing server-side preservation, the materials required to run the server should be archived, which is likely to require close collaboration with IT staff or whoever is responsible for those servers. The key aspects to be agreed are:

- Levels of access and security permissions for conservators and web developers, to allow them to document the artwork on the server, as well as upload, download and/or modify the software and other components that build the website.

- Notifications of any updates to the web server software environment that might impact the artwork. Web servers often host many websites so decisions about which version to use for the operating system, databases, programming languages or other software requirements could be made on the basis of many users' needs and thus unintentionally break a web-based artwork. Timely notification allows conservators to test the artwork and ensure that there was no change.

- Ensuring the security of the web server, particularly with respect to network and systems integrity, to avoid malicious tampering with the artwork. If there is visitor input, the hosting institution will need to consider monitoring the input for offensive content and agree on how and whether to respond.

The conservator should document the hierarchy or "file tree" of the files for the artwork on the web server, the specific programming languages and dependencies and any specific network considerations along with standard web server issues such as security, storage, backing up and server maintenance.

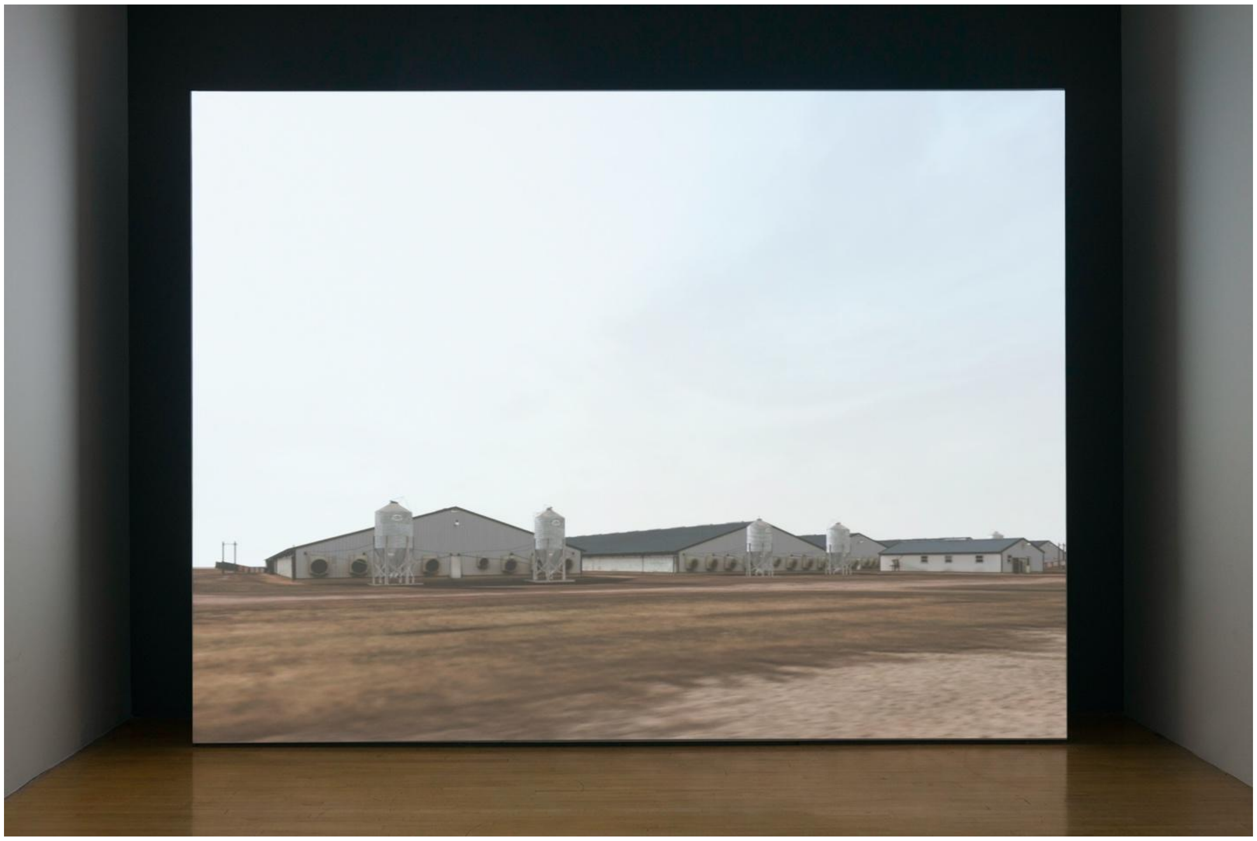
It is important to test all web-based artworks regularly to document and address any problems that may arise. Using a variety of current browsers on a scheduled basis to check for anomalies or problems helps to ensure cross-browser compatibility or to highlight the need to advise users which browser is preferred if necessary.

For web artworks where server-side preservation is not possible, or as a complementary measure, client-side preservation measures, sometimes referred to as web archiving, may be an option (National Archives 2011). Specialized tools such as web crawlers can be used to retrieve the code and data displayed by a browser in the form of a WARC (WebARChive file format), a standardized web archiving format (Library of Congress - WARC 2020). These recordings maintain some navigation functionality among the pages of the site, and a degree of interactivity. Tools such as Conifer (Rhizome - Conifer (software) 2020) and Webrecorder (Webrecorder, n.d.) allow for small scale, user-guided recording using an intuitive interface, as well as playback of the WARC files. Web crawling tools such as Browsertrix Crawler (Browsertrix (software) n.d.), Heretrix (Heritrix3 (software n.d.) and Wget (Wget (software) n.d.) can be used to carry out automated capture of larger or more complex sites. Using client-side tools means that processes happening on the server-side are not captured, so not all recordings are successful. These tools can also be combined with server-side preservation, for instance when links to external pages are important for the work, so they can be captured preemptively.

Real-Time 3D, Virtual Reality and Augmented Reality

Real-time 3D (RT3D) rendering is the use of software to generate a moving image sequence in real-time, from 3D data sources. In contrast to pre-rendered 3D and digital video, where a fixed number of frames of image data are stored, in real-time 3D frames are generated on the fly. This allows for a moving image that is dynamic and can respond to user interaction. The development of RT3D rendering has been led by the video game industry, where the technology has become ubiquitous. Software-based artworks utilizing RT3D rendering technologies are becoming increasingly common; see for example the artists John Gerrard (see fig. 22.11) and Ian Cheng who are represented in a number of museum collections.

<<INSERT FIGURE 22.11 HERE [
TBM_Chapter_22_Software_Figure_11_Gerrard.jpg] Caption:
Figure 22.11 John Gerrard, *Sow Farm (near Libbey, Oklahoma) 2009* (2009), installation at Tate Britain in 2016. This software-based artwork uses real-time 3D rendering to create a virtual representation of a remote pig farm in the USA.



RT3D software is developed with the assistance of existing libraries and tools. Perhaps the most significant of these is what is known as an *engine*. An engine is software which packages a set of reusable features and tools for RT3D software development. This will typically include a development environment, known as an editor, which incorporates WYSIWYG elements and graphical interfaces that simplify the process of development for non-programmers. Within this development environment, 3D scenes can be built from assets imported into the engine (such as 3D meshes, textures and audio), while interactivity and dynamic events can be added through scripting languages or visual editors. Engines are created using high-level programming languages like C++, and can be extended by users through plugins, or the modification and recompilation from source (if the developer makes this available). Prominent examples of RT3D engines include Unity and Unreal Engine.

Using an engine editor, a user can build executable software to run on a variety of different platforms. This consists of a package containing all of the necessary code and data to run the software on the target platform — usually a particular operating system and a 3D API (e.g., DirectX or OpenGL). Real-time 3D software often has a close link with dedicated graphics hardware, which is designed to offload common calculations in 3D rendering from the CPU. The presence of a GPU, alongside other powerful hardware components, ensures that frames can be generated at a rate that is sufficient to maintain an illusion of motion in the images displayed on an output device. This can be measured using the metric *frame rate* (the number of frames generated per second or FPS) or *frame time* (the amount of time in milliseconds taken to render a frame). The GPU hardware, drivers and the 3D API used may alter the characteristics of a rendered image.

Recent research on the preservation of real-time 3D artworks has focused on virtual reality (VR) artworks (Campbell and Hellar 2019; Ensom and McConchie 2021; Brum et al. 2021) and augmented reality (AR) artworks (Fortunato and Heinen 2020). VR and AR technologies, sometimes referred to using umbrella terms such as immersive media and XR, build on real-time 3D rendering. These technologies are centered on hardware technologies that increase levels of immersion in the virtual environment. VR and AR differ in the extent to which they immerse the user in a virtual environment. For VR, the user only sees the virtual environment, whereas AR uses cameras mounted on an HMD or mobile device to display the real-world, which is then augmented with virtual elements. Mixed reality (MR) artworks combine elements of VR and AR.

Common immersive media technologies include the head-mounted display (HMD), a display device that mounts one or two screens in front of a user's eyes, providing a wide-field of view. An HMD can achieve a stereoscopic image by rendering two frames at slightly different viewing angles, one for each of the user's eyes which creates the illusion of depth. A high frame rate must be achieved to prevent motion sickness in the user, which currently requires the use of a powerful computer system and application of frame interpolation techniques. In addition to HMD technology, VR and AR also make use of tracking systems. This allows the real-time 3D software to track the user’s position in the physical environment and translate this into movement within the virtual environment.

Physical Computing

Physical computing, in the context of software-based art, usually refers to projects of a DIY nature, that use combinations of low-cost microcontrollers (e.g. Arduinos) or computers (e.g., Raspberry Pi) with sensors and other electronic components to control output devices. These components are often used to generate interaction or respond to external inputs. They can be used, for instance, to read inputs such as light on a sensor, a finger on a button, or a Twitter message, and turn them into an output such as activating a motor, turning on an LED, or publishing content online (Arduino (website) n.d.).

This type of technology is closely aligned with open-source communities and hacker practices, and the software tools are often open source and hardware specifications openly shared. Physical computing is part of the curriculum in many art schools, as a fun way to introduce programming concepts. The robustness of the components and movable parts are usually the main preservation concern (see also 22.3.1), and although these components tend to be generic in the functionality they offer, they may have an aesthetic value and may be difficult to replace in the future.

3D-Printed Art

The 2021 conference *TechFocus IV - Caring for 3D-printed Art* described 3D-printed art as “the process of creating a three-dimensional object via computer-aided design (CAD) programs and digital files, printing it using a range of materials from plastic to metal more conventionally, to all kinds of experimental materials like chocolate or shrimp shells." (AIC - TechFocus IV 2021). 3D printing is also known as *additive manufacture* and *rapid prototyping*. While the artworks produced using 3D printing technologies are generally cared for by object conservators, preserving the software and technology required to create these artworks may also provide a role for time-based media conservators as more artworks using this medium enter collections and software obsolescence is seen as a risk factor. TBM conservators may thus encounter .stl (standard tessellation) files which express triangular representations of 3D objects (McCue 2019). Artists "may also be willing to supply the working design files, in Computer-Aided Design (CAD), Rhino©, or other software" (Hamilton 2017).

The fragile nature of the objects themselves may necessitate decisions on the materiality and conservation of the objects in cases where a museum and the artist make decisions about re-printing the object (Oleksik and Randall 2018). Objects conservator Emily Hamilton has observed that "Some artists feel more strongly about the prints, while for others the files are the art" (Hamilton 2020). Collaboration between object conservators and TBM conservators will best support the conservation of these artworks.

On the acquisition of .stl files at MoMA:

At MoMA, Media and Sculpture Conservation collaborated on a case study of the artwork *Altar/Engine* (2015) by Tauba Auerbach. Consisting of over 40+ 3D printed objects, the artwork also arrived at acquisition with the .stl files used to print the objects and a composite .stl file of the entire piece in a 3D environment. Media and Sculpture Conservation reviewed the composite .stl file to help guide us in setting installation parameters for the piece (how the objects are placed on a table surface and the distance between them, for example) but quickly realized that the .stl files used to generate the objects were also a critical resource as it allowed us to evaluate the condition of the objects and asses how much they have degraded over time. A number of the objects, it was observed, had already started to collapse on themselves which we only realized in this comparison. Additionally, we used the .stl files to aid in designing the packing for long-term storage for some objects to provide better support. The use of .stl files as documentation, reference and as a resource has since become a critical element of our long-term strategy for caring for these objects and we now regularly request them when accessioning 3D printed/rapid prototype objects.

Peter Oleksik, Associate Media Conservator, Museum of Modern Art, New York City. (Oleksik 2021)

22.4 Preservation and Treatment of Software-based Art

In the long-term care of software-based art, we must contend with the inevitable failure and obsolescence of components (see 22.3.3 and 22.3.7). In order to continue to display software-based art, it may therefore be necessary to make changes to the system used to realize the artwork. Through carefully considered modification of components, we can extend the life of the artwork without compromising its identity. Despite the need to accommodate change in the conservation of software-based art, the technologies with which a work was created will always be an important part of the artwork's history, reflecting the period and context of creation, as well as the artist's practice. They must therefore always be thoroughly documented, even if they are replaced to ensure the work remains displayable.

In this section we present a variety of approaches to preservation, treatment, and interventions for computer- and software-based art including both environment and code interventions. Code intervention is the modification of code in order to maintain the functionality of an artwork. Environment intervention strategies contrast with code intervention strategies in that they involve modification to the hardware or software environment surrounding the artwork. In practice, you may wish to choose one of these intervention approaches or combine them as dictated by the requirements of the treatment.

22.4.1 Introduction to Environment Intervention Strategies

Environment intervention strategies are those which alter or adapt the technical environment in which software runs, so that access to that software can be maintained. A technical environment consists of two parts: the *hardware environment*, which is made up of physical hardware components; and the *software environment*, which consists of off-the-shelf software components (as opposed to artist-created software). Environment intervention strategies can be used to address a wide variety of problems in the care of software-based art, such as the failure of a hardware component, and allowing software to run on computers for which it was not designed. In this section, we will introduce various approaches which can be taken, their limitations and some general principles for their application.

22.4.2 Environment Maintenance

Environment maintenance is the process of keeping a technical environment operational, so that software can continue to be run within this environment. This can include processes such as repairing hardware and updating off-the-shelf software. This approach is most likely to be effective in the short term, when access to replacement hardware components, software updates and the expertise required to use them remains available. As maintenance continues over extended periods of time, compatible hardware components may become increasingly difficult to find, updates to off-the-shelf software may cease and expertise in the technologies may dwindle or disappear altogether. When such factors become limiting, environment migration, environment emulation or code intervention strategies should be explored.

One common example of environment maintenance is the replacement of failed hardware components. For example, if an internal component of a computer fails, it could be replaced with an identical or similar device. Similarity can be very important when making such replacements, as this will help lower the risk of breaking dependency relationships. Important factors to consider include the performance of replacement components, interface between the component and motherboard and the availability of drivers for the target OS. For example, it may not be possible to replace an IDE/PATA hard drive with a SATA hard drive unless this newer interface technology is supported by the motherboard and operating system.

Maintenance can go beyond replacement of components and address failure directly through the repair of electronics at the circuit level. Successfully doing so is contingent on suitable electronic components being available, and access to the tools and expertise required to carry out the replacement. For example, a specialist might be able to remove a failed capacitor, identify a suitable replacement by carefully matching the specification, and solder it into place in the circuit. However, this kind of repair can be very challenging (if not impossible) for newer computer hardware due the use of industrially manufactured PCBs which can be high density, multi-layered and use components which are not widely available (Stricot and Vlaminck 2021).

Off-the-shelf software updates are another form of environment maintenance; for example, applying security updates to an operating system or updating to a newer more stable hardware driver. As a consequence of dependency relationships, and the alteration or removal of features by software developers over time, carrying out this kind of update can risk loss or change of work-defining characteristics. For networked artworks, where public internet access must be maintained over extended periods of time, security risks are an added concern. The software on a server supporting a web artwork will need to either be updated, or the server securely emulated (see 22.4.4). It is important to note that, if the environment is updated, the artist-created source code must be tested for compatibility and evaluated for possible code intervention (see 22.4.7).

22.4.3 Environment Migration

Environment migration is the process of moving software from one technical environment to another newer technical environment without altering the software itself. An example of this approach would be installing software designed for an obsolete computer system on a contemporary computer system. Where possible, this is advantageous in the conservation of software-based artworks, as it is one way of mitigating the risks posed by the failure and obsolescence of components. However, its usefulness is often limited by the tendency for technical environments to change over time, which can break dependency relationships between the software you wish to preserve and its technical environment. For example, software designed to run on the Macintosh computers of the 1990s and early 2000s, which used PowerPC CPUs, will no longer run on the Intel-based and ARM-based Macintosh computers available today. For this reason, environment migration tends to be feasible only in a relatively short-term timeframe.

Assessing the viability of environment migration should be led by an understanding of the dependencies of the software at the center of the software-based artwork system. For example, in the case of replacing an aging computer system with a newer one, information such as the required CPU architecture and operating system will help determine a suitable system to migrate to. Ideally, these two elements of the hardware specification would be matched in the newer computer system. However, due to the process of obsolescence this may be difficult to achieve. For example, as Windows XP is no longer sold or supported by Microsoft, hardware drivers are rarely created to support modern hardware components on this operating system. A change in the hardware environment may therefore precipitate a change to the software environment. For software developed to run on Windows XP, there is a chance of success depending on whether it uses any elements of Windows XP which are no longer present in a newer version of Windows such as Windows 10.

In the case of the artwork *Subtitled Public* (2005) by Rafael Lozano-Hemmer, it was possible for conservators at Tate to install and run the Windows XP software on Windows 10 without altering it. However, a quirk of the programming caused the software, which carries out motion tracking using a computer vision system, to run with significantly increased latency. The source code was refactored and recompiled by the artist's programmer to resolve this. Further risks that may require future intervention were identified as part of the treatment; most significantly that a Windows component used by the software to access video camera feeds has been deprecated — meaning its use is discouraged by Microsoft — and may be removed in future versions of Windows. As this example illustrates, environment migration is often a short-term measure where it is possible at all. In situations where there have been significant changes in prevalent hardware and software environments, such as a CPU architecture change, environment emulation may be an effective alternative.

22.4.4 Environment Emulation

Rather than moving software to a new technical environment, it may be more advantageous to *emulate* a suitable environment. Emulation can be defined as the use of software or hardware tools called emulators, which allow one computer system to behave as if it were a different computer system. Emulation is useful because it allows us to run operating systems and other software on computer hardware which they were not designed for. This has significant advantages in supporting long-term access to digital materials broadly, as emulating an obsolete computing platform allows us to maintain access to it without requiring physical hardware, which as discussed above, becomes harder to repair and source over time. Emulation was first widely discussed in a digital preservation context after a series of articles by Jeff Rothenburg in the 1990s (Rothenberg 1995; Rothenberg 1999). Its value as a preservation strategy is now supported by a growing body of evidence from a variety of digital preservation contexts (Wheatley 2004; von Suchodoletz et al. 2008; Espenschied et al. 2013; Dietrich et al. 2016).

The term emulation can be used to refer to a number of related approaches. Emulation in its strictest sense — known as *full-system emulation* — is where all hardware is completely simulated within the emulator. At time of writing, software tools are the best option for emulating desktop computers and will be the focus of this section. However, hardware devices are also available, such as field-programmable gate array (FPGA) integrated circuits that can be programmed to mimic other hardware. When talking about emulation using a software emulator, the physical computer system can be referred to as the *host*, while the computer system that is emulated in software can be referred to as the *guest*.

On Emulation-as-a-Service

While local access — that is, access via a physical computer at a specific location — is useful for the display of software-based art in physical exhibition spaces, it is not the only way to provide access to emulated software-based art. We have recently seen the emergence of platforms that provide web browser-based access to emulators running on remote servers (von Suchodoletz et al. 2013; Liebetraut et al. 2014). This approach is known as *emulation-as-a-service* (EaaS) and has already been applied to providing access to software-based art from the Rhizome collection (Espenschied et al. 2013). The Scaling Emulation as a Service Infrastructure (EaaSI) project is continuing the development of the EaaS software and infrastructure (Software Preservation Network - EaaSI n.d.).

The great advantage of the EaaS approach is that it allows the presentation of emulated software-based art to anyone with an internet connection and a web browser. It also greatly simplifies the process of using emulators by shifting emphasis to external management of the technical aspects of emulation through a central location, and then providing access to pre-configured environments to suit a variety of common use cases. The disadvantage to this is that control of the technical aspects of emulation is handed over to the service provider which may reduce configuration and customization options. EaaS tends to be most successful for software-based artworks in which the software has few external dependencies (e.g., on physical peripherals) and requires a single video output. It has been successfully used for emulating web-based artworks, where it can offer a secure way of providing public access to web servers that require outdated software (Roeck et al. 2019).

Virtualization

Virtualization is similar to emulation but differs in that it allows the guest system access to some of the underlying guest hardware. This is most frequently encountered in CPU virtualization, in which a specialized piece of software called a hypervisor allows the virtualization software to execute code directly on the physical processor of the host machine. The condition of this is that the guest operating system is able to make use of the same instruction set architecture (ISA) as the host. This is because unlike emulation, processor instructions are not translated to suit execution on the host processor. This has significant advantages for the performance of the guest, as having direct access to underlying hardware allows the software to execute at closer to native speeds than would be possible when instructions are translated in full system emulation. Virtualization has been demonstrated as a viable strategy in the preservation of software-based art (Lurk 2008; Falcão et al. 2014).

While the term virtualization is often used synonymously with CPU virtualization, other components may also be virtualized. This is used for GPUs or other components where it is desirable for them to be shared between guest and host, and where it is typically known as *paravirtualization* (Roeck 2018). A related technique called *passthrough* may also be used to allow guests direct access to host hardware components (e.g., a USB device). While this allows it to be used at native speed, this comes with the disadvantage that the hardware component cannot be shared with the host. This also affects preservation prospects, as virtualized components are more closely tied to physical hardware and the host operating system-specific drivers to support them (Rechert et al. 2016).

Containerization

Containerization (also known as *operating system -level virtualization*) involves a similar paradigm to virtualization but instead relies on *containers*: packages of software, consisting of the program to be run and any software dependencies it requires to run on a host operating system (IBM Cloud Education 2021). This container is stored in a form that is easily shared and reused with minimal configuration required in order to use it, known as a container image. A container image can conform to a variety of file format specifications. Much like virtual machines, containers have found use in the context of web development, where they offer a means of running multiple software environments on the same server hardware. Unlike virtual machines, container images do not contain an operating system. As a result, they are smaller in size and rely on a container runtime, such as Docker Engine or containerd (Cloud Native Computing Foundation n.d.), instead of a hypervisor (see Virtualization). Containerization can achieve faster performance than emulation or virtualization.

Containerization is finding use in the presentation of web-based artworks. For example, it has been used as a way to provide access to old versions of web browsers by Rhizome (Rossenova 2020a). It is also used by Haus der Elektronischen Künste in Basel for managing web servers, which Claudia Roeck describes as follows:

> "Each artwork is in a container, and each container contains a web server with the artwork. This is because they all need different scripting language versions, database versions, everything — each artwork has a different environment. Some might use the same software, but they're not using the same versions, so you can't really have them on the same web server. So, for each work you need a web server, and they are packed each in one container." (Roeck 2020)

Containerization does not have the preservation applications of emulation or virtualization however, which achieve a level of independence from the physical hardware (Roeck 2020). Containerization is instead reliant on an additional piece of off-the-shelf software (the container engine) and any software packaged in a container must be able to run on the host operating system.

22.4.5 Choosing an Environment Intervention Approach

When choosing which environment intervention approach to apply to a work requiring treatment, the first step should be to identify whether hardware can be repaired or replaced like-for-like. If so, environment maintenance is likely to be a good option to explore, as this minimizes risk of unintended alterations to the artwork. If repair or replacement is not an option, or is prohibitively costly, you can assess whether or not carrying out environment migration is likely to be feasible. Decisions about how to approach this must be made on a case-by-case basis but will usually involve assessing the dependencies of the software and determining whether or not they can be maintained after the proposed migration process. This may necessitate a certain amount of experimentation, as it is not always possible to predict the impact changes will have on the execution of the software. When it is not possible to migrate to newer hardware, an intervention involving environment emulation should be considered.

22.4.6 Emulation in Practice

When deciding which approach to emulation to take, consider:

- What emulation tools are available to support access to this kind of software on modern computers?
- What are the hardware dependencies of the software, and can one approach support these better than another?
- What are the performance requirements of the software, and can one approach support these better than another?
- To what extent does this approach mitigate obsolescence and how soon would another intervention be required?
- Will this approach support requirements for access to and/or display of the software-based artwork?

Experimentation with different tools may be necessary to find the approach that best meets the aim of the intervention. Time-based Media Conservator Jonathan Farbowitz describes a typical emulation use case as, “a very simple setup of software running on a single computer with a single video output, and a minimum of peripherals” (Farbowitz 2020). More complex use cases can come up against the limitations of emulation technology (e.g., works requiring 3D acceleration capabilities or access to peripherals).

There are a large number of tools and workflows for carrying out emulation, and we do not cover these exhaustively here. Instead, we present a simple workflow that will help make applying emulation successful. This sequence of steps is informed by the workflow in use at Tate, which was developed during the Software-based Art Preservation Project (Ensom et al. 2021), based on outcomes of a research collaboration between Klaus Rechert and Tate in 2016 (Rechert et al. 2016).

1. **Select an emulator.** Use information about the original computer system on which the software was designed to run (e.g., processor architecture, operating system) to select a suitable emulator (see Table 22.3). Suggested open-source emulators are Basilisk II (Bauer and Various n.d.), PCem (Walker 2007), QEMU (The QEMU Project Developers 2010), SheepShaver (Bauer and Various n.d.) and VirtualBox (Oracle Corporation n.d.). Choosing an emulator which also supports virtualization can improve performance.
2. **Prepare a disk image.** A disk image (see Chapter 14) will act as a storage device for the emulated computer system. If the disk image has been acquired from a physical machine, you may need to convert the image to a format supported by that emulator. If you have not acquired a disk image from a physical machine, you can create a synthetic disk image (see Chapter 14 and Espenschied 2017). This is an empty disk image container and will require you to install an operating system, and any other software required, via the emulator. Two useful tools for creating and converting disk images are qemu-img (

QEMU Project Developers 2021) associated with the QEMU emulator and VBoxManage (Oracle Corporation 2004) associated with the VirtualBox emulator.

3. **Configure the emulator.** Options within the emulator should be used to match or approximate the hardware requirements of the software. Ensuring these are as closely matched as possible increases the chance of success.
4. **Run the emulation.** If you experience problems at this step (e.g., the disk image does not boot), you can experiment with returning to the previous steps and modifying the emulator configuration or the disk image.
5. **Assess the emulation.** Assess the success of the emulation by making use of side-by-side comparison with physical hardware, image and video reference materials, quantitative metrics and other documentation. See Magnin (2015) and Ensom (2018) for example approaches.
6. **Document the emulation.** Document the process so that it can be understood in the future. Important information to record includes the workstation used, the versions of software tools used, any modifications made to the base disk image (e.g., format conversion), a record of the configuration options used with the emulator and notes on the success of the emulation (e.g., problems encountered and solutions developed, differences between the emulation and the original hardware).
7. **Archive the emulation.** Determine whether you wish to archive the components of the emulation so that it can be repeated in the future. Where emulation was successful, and particularly where the emulation was used to display the work, doing will help support future treatment and display. Archived components should include the binaries or installer for the specific version of the emulator and the disk image used, along with documentation as described in the previous step.

Table 22.3 Summary of commonly encountered processor architectures, associated operating systems and a selection of actively developed emulators capable of running them.

Processor Architecture	Associated Operating System(s)	Suggested Emulators
Motorola 68000 (also known as 68k)	Mac System 7	Basilisk II QEMU (qemu-system-m68k)
Intel 80386 (also known as i386, IA-36 or x86-32)	Windows 9x (95, 98, Me, 2000) Windows XP Windows Vista Ubuntu	PCem QEMU (qemu-system-i386) VirtualBox
PowerPC (also known as ppc)	Mac OS 8 Mac OS 9 Mac OS X (10.0 - 10.5)	QEMU (qemu-system-ppc) SheepShaver
x86-64 (also known as AMD64 or Intel x86-64)	Windows Vista Windows 7 Windows 8	QEMU (qemu-system-x86_64) VirtualBox

	Windows 8.1 Windows 10 Mac OS X (10.4 or later) Ubuntu	
ARM	MacOS 11 or later Ubuntu (and other Linux distributions) Android iOS	QEMU (qemu-system-aarch64) [limited or no support for some operating systems e.g., iOS]

Emulation is often an iterative process of troubleshooting problems through the testing of different tools and configurations, and consulting of existing online resources where these may well have been encountered before. Some useful practical resources include the E-Maculation forum (Emaculation.com n.d.) for Mac emulation, the QEMU QED website (Gates n.d.) for guidance on using the QEMU emulator, and the developer-created user documentation for the tools mentioned above. Although not every emulation attempt will be successful, emulation and other forms of environment intervention are also opportunities to gain knowledge about your software-based artwork; by testing different intervention approaches, we learn about and document the connections between a piece of software and its technical environment.

22.4.7 Introduction to Code Intervention

In some cases, code intervention — the modification of code in order to maintain access to software — is deemed necessary to restore a work of software-based art. Evaluating the scale of intervention needed is a first step towards remediation. For example, a relatively small intervention of one or two lines of code would be required to modify the source code of a scripting language when an update is required to access a different URL linking to an external dependency such as a data source or media player. Additional steps would be required even for this small change if the software for the artwork needs to be recompiled. If more substantial changes are required, for example where an obsolete version of a programming language is no longer supported, the code would need to be migrated to another programming language.

22.4.8 Code Migration

Migration is a term that has different meanings in a variety of contexts. With respect to conservation of film and video, it usually refers to a change in the carrier or format of the media. For software-based artworks, it can refer to migrating an artwork from one hardware environment to another for exhibition or preservation purposes (see 22.4.3). *Code migration* refers to the migration from one programming language or development environment to another and is an intervention that requires a review and re-writing of parts or most of the source code.

Migrating a software-based work of art from one programming language or development environment to another can be a resource-intensive process, undertaken after considering “the existing migration options [that] are useful for preserving software-based art and how they compare to the emulation options currently used" (Roeck et al. 2019). When evaluating the migration of a work of software-based art from one programming language or development environment to another, there are a number of considerations that are pertinent to software-based artworks that are not necessarily relevant to other software applications. This is due to the conservation goals of minimal intervention, significance of the artist-authored code, respecting reversibility, and selecting programming solutions that best support long-term preservation of the artwork.

Selecting the Target Programming Language

Following are some of the questions to consider when selecting the programming language to migrate the artwork to. First, is there a newer version of the same programming language or development environment available? If so, then upgrading to a newer version of the software or environment that the artist originally selected is in keeping with the artist's choice of language or environment at the time that the work was created, taking care to check for missing or new features that are not backward-compatible. In addition, migrating a software application to a newer version of the same language or environment is likely a well-documented process and it is possible that some of the migration can be handled programmatically either by using custom software written for this purpose or reviewing tools available online. For example, software has been written to do a rough "first pass" on converting Python 2.x code to Python 3.x code.

If this is not an option, the next question to address is whether there is another programming language that is similar in syntax and structure that could be used. While making decisions around migration, other considerations play a role as well. For example, are there analogous functions, libraries, or other code sources to support the restoration as envisioned? As artists make use of third-party libraries for graphics, media players, special effects and other purposes, it is important to ensure that these libraries are compatible, available in the new programming language or environment or that comparable libraries are available. Conservators along with their IT or programmer specialists should also consider how and whether the software and hardware environments for the specific artwork would need to be configured to accommodate the migration and if so, whether that would cause further issues. Finally, as is true for all software, a determination needs to be made that the language selection will support all of the required systems and functional requirements to render the artwork's behaviors as intended. In the case of software-based artworks, special attention must be paid to how the artwork will appear to ensure visual and artistic integrity (which is not necessarily true for software application migrations outside of the visual arts). Long-term preservation of the new programming language is of great importance so that future interventions can be postponed as long as reasonably possible.

Code Resituation

When a new programming language must be selected, it is important to address whether there is another programming language that is similar in syntax and structure that could be used. For example, when restoring a web-based work of art that runs as a Java Applet, the intervention will require migration to another programming language as Java Applets no longer run under standard browsers. In such a case, one could consider Python or JavaScript, to name two examples; between the two, the syntax and structure of JavaScript is much closer to Java than

Python, so although a Python restoration could ultimately look the same and behave the same as the original artwork, the code would by necessity look quite different.

With Java and JavaScript on the other hand, some portions of the artist's original code can be copied as-is from Java to JavaScript, along with a small library of functions to manage some of the differences between the languages (such as the way that floating point numbers are handled). This approach was developed at the Solomon R. Guggenheim Museum during the restoration of John F. Simon Jr.'s web artwork *Unfolding Object* (2002) (Phillips et al. 2018) and called *Code Resituation* (Engel and Phillips 2018). In this case a significant amount of *Unfolding Object's* original code could be retained, including the algorithms that render the abstract "objects" when users interact with the work, thus ensuring fidelity to the artist's work. Further, when code can be migrated in this way, aspects of the functionality are therefore not affected which in turn supports testing and A-B comparisons during development.

Code Migration: Quality Control and Documentation

As with other treatments, it is important to conduct standard software development testing throughout development to ensure performance. Web-based artworks should also be tested using all of the commonly used browsers and on different platforms (Windows, MacOS and Linux, if available). Appropriate "A-B comparisons" running the work in its original form if that can still be arranged alongside the restored version can be part of the quality control process throughout the development process as needed. For web-based artworks, a development environment (e.g., a website or platform such as AWS that is configured to simulate the institutional or collection web server but not available to the public) is valuable throughout restoration treatment and testing. During this phase, the programmer(s), conservator, and the artist, if appropriate, can all have access to view the progress. A VCS implementation such as git is helpful in tracking progress, bugs, and documenting the software migration. Upon completion of the restoration, it is advisable to download a report from the VCS software to capture all of the steps and decisions made throughout the process as part of the treatment documentation or as a separate treatment report. Any code intervention should also be documented within the source code of the restoration copy through code annotation (for examples, see Phillips et al. 2017; Engel and Phillips 2019).

22.5 Conclusion

Hardware and software will continue to change as new devices, new programming languages and new paradigms become available over time; a comprehensive list of scenarios is not possible, nor would we want it to be as artists continue to explore new technological horizons in their work. Those caring for works of software- and computer-based art will be faced with languages and environments that are not discussed in this chapter as they have not yet been developed. However, the principles remain the same: to understand the technologies behind the hardware and software and the possible impacts of failure and obsolescence; to consider the nature of the programming language(s) and other tools used in creating the artwork; and the role of data, media files and external dependencies, if any. Some emerging challenges for conservators will include the ongoing introduction of industrial algorithmic processes such as artificial intelligence and machine learning into artists' works, and technical and ethical implications of NFTs and other blockchain technologies. Breaking down these new artworks into their respective components, the “anatomy of the artwork” as it were, will inform the decision-making of conservation teams moving forward into the future.

Conservation treatment options will evolve, as the time-based media conservation community continues to pioneer new intervention approaches to help safeguard the longevity of software- and computer-based art. Our response to new technologies and challenges will be best supported through collaboration with colleagues in the wider digital preservation and computer science communities.

Acknowledgements

The authors wish to thank Lan Linh Merli-Nguyen Hoai, Time-based Media Conservation Fellow at Düsseldorf Conservation Center and Professor Craig Kapp, Clinical Professor of Computer Science, Department of Computer Science, Courant Institute of Mathematical Sciences at New York University for taking time to read this chapter and provide valuable feedback.

BIBLIOGRAPHY

AIC (American Institute for Conservation). “TechFocus III: Caring for Software-Based Art.” American Institute for Conservation (AIC) and Foundation for Advancement in Conservation (FAIC) Resource Hub, September 25, 2015. https://resources.culturalheritage.org/techfocus/techfocus-iii-caring-for-computer-based-art-software-tw-2/.

———. “TechFocus IV: Caring for 3D-Printed Art.” AIC (American Institute for Conservation), 2021. https://learning.culturalheritage.org/products/techfocus-iv-caring-for-3d-printed-art.

Archive of Digital Art (ADA). “ADA | Archive of Digital Art (Website).” Accessed February 27, 2021. https://www.digitalartarchive.at/nc/home.html.

Arden, Sasha, and Mark Hellar. Interview with Sasha Arden and Mark Hellar by Patricia Falco, Tom Ensom and Deena Engel, October 26, 2020.

Arduino. “Arduino (Website),” 2022. https://www.arduino.cc/.

Association of Research Libraries (ARL). “Code of Best Practices in Fair Use for Software Preservation,” 2019. https://www.arl.org/wp-content/uploads/2018/09/2019.2.28-software-preservation-code-revised.pdf.

Barok, Dušan, Julie Boschat Thorez, Annet Dekker, David Gauthier, and Claudia Roeck. “Archiving Complex Digital Artworks.” *Journal for the American Institute of Conservation* 42, no. 2 (2019a): 94–113. https://doi.org/10.1080/19455224.2019.1604398.

Bauer, Christian, and Various. *Basilisk II (Software)*. Accessed December 20, 2021. https://basilisk.cebix.net/.

———. *SheepShaver (Software)*. Accessed December 20, 2021. https://sheepshaver.cebix.net/.

Bavota, Gabriele, Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Marquez, Mario Linares-Vasquez, Laura Moreno, and Michele Lanza. “Software Documentation Issues Unveiled.” In *2019*

IEEE/ACM 41st International Conference on Software Engineering (ICSE), 1199–1210. Montreal, QC, Canada: IEEE, 2019. https://doi.org/10.1109/ICSE.2019.00122.

Brokerhof, Agnes W. "Installation Art Subjected to Installation Art Subjected to Risk Assessment – Jeffrey Shaw's 'Revolution' as Case Study." In *Inside Installations*, edited by Tatja Scholte and Glenn Wharton, 91–101. Amsterdam University Press, 2011. https://www.academia.edu/8034919/Installation_Art_Subjected_to_Installation_Art_Subjected_to_Risk_Assessment_Jeffrey_Shaws_Revolution_as_Case_Study.

Browsertrix Crawler (Software). Rhizome Webrecorder. Accessed February 7, 2022. https://github.com/webrecorder/browsertrix-crawler.

Brum, Olivia, Mauricio van der Maesen de Sombreff, Eléonore Bernard, and Gaby Wijers. "A Practical Research into Preservation Strategies for VR Artworks on the Basis of Justin Zijlstra's 100 Jaar Vrouwenkiesrecht." LIMA and the Dutch Digital Heritage Network, 2021. https://www.li-ma.nl/lima/sites/default/files/Rapport_A-Practical-Research-into-Preservation-Strategies-for-VR-artworks-on-the-basis-of-Justin-Zijlstras-100-Jaar-Vrouwenkiesrecht.pdf.

Bunz, Sophie. "Preserving Stephan von Huene's Electronic Artworks by Means of Bit-Stream Documentation." *The Electronic Media Review* Five: 2017-2018 (2018). https://resources.culturalheritage.org/emg-review/volume-5-2017-2018/bunz/.

Campbell, Savannah, and Mark Hellar. "From Immersion to Acquisition: An Overview Of Virtual Reality For Time Based Media Conservators." *Electronic Media Review* Volume Six: 2019-2020 (2019). https://resources.culturalheritage.org/emg-review/volume-6-2019-2020/campbell/.

Cloud Native Computing Foundation. *Containerd (Software)*. Go. Accessed December 20, 2021. https://containerd.io/.

Cranmer, Candice, and Nick Robinson. Interview with Candice Cranmer and Nick Robinson by Tom Ensom. Audio Recording, March 3, 2021.

Creative Commons (CC). "Creative Commons - About the Licenses." Accessed December 1, 2021. https://creativecommons.org/licenses/.

Dekker, Annet. *Collecting and Conserving Net Art: Moving beyond Conventional Methods*. London ; New York: Routledge, Taylor & Francis Group, 2018.

Dekker, Annet, and Patricia Falcão. "Interdisciplinary Discussions about the Conservation of Software-Based Art Community of Practice on Software-Based Art." London: Tate, PERICLES, 2017. https://openresearch.lsbu.ac.uk/item/8v21z.

Della Casa, Davide. "PDFs Of 'Computer Graphics And Art.'" *TopLap* (blog), September 27, 2012. https://toplap.org/pdfs-of-computer-graphics-and-art/.

Di Cosmo, Roberto, and Stefano Zacchiroli. "Software Heritage: Why and How to Preserve Software Source Code." In *IPRES 2017 - 14th International Conference on Digital Preservation*, 1–10. Kyoto, Japan, 2017. https://www.softwareheritage.org/wp-content/uploads/2020/01/ipres-2017-swh.pdf.

Dietrich, Dianne, Julia Kim, Morgan McKeehan, and Alison Rhonemus. "How to Party Like It's 1999: Emulation for Everyone." *The Code4Lib Journal*, no. 32 (April 25, 2016). https://journal.code4lib.org/articles/11386.

Digital Preservation Coalition (DPC). "What Are the Risks?," 2022. https://www.dpconline.org/digipres/implement-digipres/dpeg-home/dpeg-risks.

Emaculation.com. "Emaculation.Com | Forum." Accessed February 1, 2022. https://www.emaculation.com/forum/.

Engel, Deena, Lauren Hinkson, Joanna Phillips, and Marion Thain. "Reconstructing Brandon (1998-1999): A Cross-Disciplinary Digital Humanities Study of Shu Lea Cheang's Early Web Artwork." *Digital Humanities Quarterly* 12, no. 2 (2018): 44. http://www.digitalhumanities.org/dhq/vol/12/2/000379/000379.html.

Engel, Deena, and Joanna Phillips. "Applying Conservation Ethics to the Examination and Treatment of Software- and Computer-Based Art." *Journal of the American Institute for Conservation* 58, no. 3 (July 3, 2019): 180–95. https://doi.org/10.1080/01971360.2019.1598124.

———. "Introducing 'Code Resituation': Applying the Concept of Minimal Intervention to the Conservation Treatment of Software-Based Art." *The Electronic Media Review* Volume Five: 2017-2018 (2018): 22. https://resources.culturalheritage.org/emg-review/volume-5-2017-2018/engel-2/.

Engel, Deena, and Glenn Wharton. "Reading between the Lines: Source Code Documentation as a Conservation Strategy for Software-Based Art." *Studies in Conservation* 59, no. 6 (November 2014): 404–15. https://doi.org/10.1179/2047058413Y.0000000115.

———. "Source Code Analysis As Technical Art History." *Journal of the American Institute for Conservation* 54, no. 2 (May 2015): 91–101. https://doi.org/10.1179/1945233015Y.0000000004.

Ensom, Tom. "Revealing Hidden Processes: Instrumentation and Reverse Engineering in the Conservation of Software-Based Art." In *Electronic Media Review*, Vol. Five: 2017-2018. Houston, Texas, USA, 2018. https://resources.culturalheritage.org/emg-review/volume-5-2017-2018/ensom/.

———. "Technical Narratives: Analysis, Description and Representation in the Conservation of Software-Based Art." Ph.D., King's College London, 2019. https://kclpure.kcl.ac.uk/portal/en/theses/technical-narratives(e01bff94-08bd-4b83-aeef-4e7d6d5b0dfc).html.

Ensom, Tom, Patricia Falcão, and Chris King. "Software-Based Art Preservation – Project | Tate," 2021. https://www.tate.org.uk/about-us/projects/software-based-art-preservation.

Ensom, Tom, and McConchie, Jack. "Preserving Virtual Reality Artworks Report (Tate)." Zenodo, August 13, 2021. https://doi.org/10.5281/ZENODO.5274102.

Espenschied, Dragan. "Emulation and Access." Museum of Modern Art (MoMA), December 8, 2017. https://vimeo.com/278042613/a78ee5e46b.

Espenschied, Dragan, K. Rechert, Dirk von Suchodoletz, Isgandar Valizada, and Nick Russler. "Large-Scale Curation and Presentation of CD-ROM Art." In *IPRES*, 2013.

Falcão, Patricia. "Developing a Risk Assessment Tool for the Conservation of Software- Based Artworks," 2010. https://www.academia.edu/6660777/Developing_a_Risk_Assessment_Tool_for_the_conservation_of_software_based_artworks_MA_Thesis.

———. "Preservation of Software-Based Art at Tate." In *Digital Art through the Looking Glass: New Strategies for Archiving, Collecting and Preserving in Digital Humanities*, edited by Oliver Grau, Janina Hoth, and Eveline Wandl-Vogt, 271–87. Krems an der Donau : [Wien]: Edition Donau-Universität Krems ; ÖAW, Austrian Academy of Science, 2019. https://www.donauuni.ac.at/dam/jcr:a29638aa-f334-4abb-9601-10e36652d09f/Digital_Art_through_the_Looking_Glass_updated.pdf.

———. “Risk Assessment as a Tool in the Conservation of Software-Based Artworks.” *EMG- Electronic Media Review* Two 2011-2012 (2011). https://resources.culturalheritage.org/emg-review/volume-two-2011-2012/falcao/.

Falcão, Patricia, Ashe Alistair, and Brian Jones. “Virtualisation as a Tool for the Conservation of Software-Based Artworks.” Melbourne, Australia, 2014. https://www.academia.edu/12462584/Virtualisation_as_a_Tool_for_the_Conservation_of_Software-Based_Artworks.

Farbowitz, Jonathan. Interview with Jonathan Farbowitz by Tom Ensom and Patricia Falcão, August 14, 2020.

Fauconnier, Sandra. “Many Faces of Wikibase: Rhizome’s Archive of Born-Digital Art and Digital Preservation.” *Wikimedia Foundation News* (blog), September 6, 2018. https://wikimediafoundation.org/news/2018/09/06/rhizome-wikibase/.

Fino-Radin, Ben. “Art In the Age of Obsolescence: Rescuing an Artwork from Crumbling Technologies.” *MoMA: Features and Perspectives on Art and Culture* (blog), December 21, 2016. https://stories.moma.org/art-in-the-age-of-obsolescence-1272f1b9b92e.

———. “Digital Preservation Practices and the Rhizome Artbase.” Rhizome at The New Museum, 2011. http://media.rhizome.org/artbase/documents/Digital-Preservation-Practices-and-the-Rhizome-ArtBase.pdf.

Fortunato, Flaminia, and Joey Heinen. “There Is No ‘I’ in IOS: Preserving Mobile Technologies as AR Experiences and Objects.” Presented at the Preserving Immersive Media Workshop, March 27, 2020. https://www.youtube.com/watch?v=6M1YgYhYGf0.

Garousi, Golara, Vahid Garousi, Mahmoud Moussavi, Guenther Ruhe, and Brian Smith. “Evaluating Usage and Quality of Technical Software Documentation: An Empirical Study.” In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering - EASE ’13*, 24. Porto de Galinhas, Brazil: ACM Press, 2013. https://doi.org/10.1145/2460999.2461003.

Gates, Ethan. “QEMU QED - The Missing Manual for Digital Preservation.” Accessed February 2, 2022. https://eaasi.gitlab.io/program_docs/qemu-qed/.

GNU Operating System. “GNU General Public License.” GNU Operating System. Accessed December 1, 2021. https://www.gnu.org/licenses/gpl-3.0.en.html.

Guggenheim Blog. “How the Guggenheim and NYU Are Conserving Computer-Based Art. Part 1.” *Guggenheim Blog* (blog), October 26, 2016. https://www.guggenheim.org/blogs/checklist/how-the-guggenheim-and-nyu-are-conserving-computer-based-art-part-1.

Guggenheim Museum. “The Conserving Computer-Based Art Initiative (CCBA).” The Guggenheim Museums and Foundation. Accessed September 2, 2020. https://www.guggenheim.org/conservation/the-conserving-computer-based-art-initiative.

Hamilton, Emily. “Email to the Authors: Book Project and a Question about 3D Printed Art,” November 16, 2020.

———. “Reprinting as a Conservation Strategy: Collecting Additive Manufactured Works at SFMOMA.” In *Future Talks 017: The Silver Edition. Visions. Innovation in Technology and*

Conservation of the Modern. Postprints of the Conference Held in Munich, 2017, edited by Tim Bechthold, 170–75, 2017.

Hellar, Mark. "The Role Of The Technical Narrative For Preserving New Media Art." *The Electronic Media Review* Volume Three 2015 (2015). http://29aqcgc1xnh17fykn459grmc-wpengine.netdna-ssl.com/emg-review/wp-content/uploads/sites/15/2018/09/EMG-Vol.-3-Hellar.pdf.

Heritrix3 (Software). Internet Archive. Accessed February 7, 2022. https://github.com/internetarchive/heritrix3.

IBM Cloud Education. "What Are Containers?," June 23, 2021. https://www.ibm.com/cloud/learn/containers.

International Centre for the Study of the Preservation and Restoration of Cultural Property (ICCROM). "A Guide to Risk Management of Cultural Heritage." Canadian Conservation Institute (CCI), 2016. https://www.iccrom.org/wp-content/uploads/Guide-to-Risk-Managment_English.pdf.

King, Chris. Interview with Chris King by Tom Ensom, 2021.

Klomp, Paul. Interview with Paul Klomp by Tom Ensom. Audio Recording, March 3, 2021.

Laurenson, Pip. "Authenticity, Change and Loss in the Conservation of Time-Based Media Installations." *Tate Papers*, no. 6 (Autumn 2006). https://www.tate.org.uk/research/publications/tate-papers/06/authenticity-change-and-loss-conservation-of-time-based-media-installations.

———. "Old Media, New Media? Significant Difference and the Conservation of Software-Based Art." In *New Collecting: Exhibiting and Audiences after New Media Art*, edited by Beryl Graham, 1st ed., 73–96. Routledge, 2014. https://doi.org/10.4324/9781315597898-4.

Library of Congress. "WARC, Web ARChive File Format." Sustainability of Digital Formats: Planning for Library of Congress Collections, 2020. https://www.loc.gov/preservation/digital/formats/fdd/fdd000236.shtml.

Liebetraut, Thomas, Klaus Rechert, Isgandar Valizada, Konrad Meier, and Dirk Von Suchodoletz. "Emulation-as-a-Service - The Past in the Cloud." In *2014 IEEE 7th International Conference on Cloud Computing*, 906–13, 2014. https://doi.org/10.1109/CLOUD.2014.124.

LIMA. "ArtHost Report 2017 - 2018." Amsterdam, The Netherlands, 2018. https://www.li-ma.nl/lima/sites/default/files/Arthost%20end%20report_FINAL%20(2).pdf.

Lintermann, Bernd, Chiara Marchini Camia, Arnaud Obermann, and Claudia Roeck. "Jeffrey Shaw, The Legible City (1989-1991)." In *Preservation of Digital Art: Theory and Practice: The Project Digital Art Conservation*, edited by Bernhard Serexhe and Eva-Maria Höllerer, 488–512. Karlsruhe: Zentrum für Kunst und Medientechnologie, 2013.

Liu, Qinyue. "An Investigation Into New Media Artists' Personal Preservation Practices." Arts Management and Technology Laboratory, December 17, 2020. https://amt-lab.org/blog/2020/10/investigation-into-new-media-artists-personal-preservation-practices.

Lozano-Hemmer, Rafael. "Best Practices for Conservation of Media Art from an Artist's Perspective." GitHub Antimodular Best-practices-for-conservation-of-media-art, September 28, 2015. https://github.com/antimodular/Best-practices-for-conservation-of-media-art.

Lurk, Tabea. "Methodologies of Multimedial Documentation and Archiving." In *Preserving and Exhibiting Media Art: Challenges and Perspectives*, edited by Julia Noordegraaf, Cosetta G. Saba, Barbara Le Maître, and Vinzenz Hediger, 149–95. Framing Film. Amsterdam: Amsterdam University Press, 2013.

———. “Virtualisation as Conservation Measure.” In *Archiving Conference*, 2008:221–25. Society for Imaging Science and Technology, 2008.

Magnin, Emilie. “Defining an Ethical Framework for Preserving Cory Arcangel’s Super Mario Clouds – Electronic Media Review.” In *Electronic Media Review*, Vol. Four (2015-2016). AIC Electronic Media Group, 2015. https://resources.culturalheritage.org/emg-review/volume-4-2015-2016/magnin/.

Malík, Martin. *HWiNFO (Software)*. Accessed October 20, 2021. https://www.hwinfo.com.

McConchie, Jack, Tom Ensom, and Louise Lawson. “Preserving Immersive Media.” Tate, 2021.

McCue, T.J. “STL Files: What They Are and How to Use Them.” Lifewire, November 14, 2019. http://guilfordfreelibrary.org/wp-content/uploads/2019/11/what-is-an-.stl-file.pdf.

McGarrigle, Conor. “Preserving Born Digital Art: Lessons From Artists’ Practice.” *New Review of Information Networking* 20, no. 1–2 (July 3, 2015): 170–78. https://doi.org/10.1080/13614576.2015.1113055.

Metropolitan Museum of Art. “Sample Documentation and Templates.” The Metropolitan Museum of Art. Accessed June 24, 2021. https://www.metmuseum.org/about-the-met/conservation-and-scientific-research/time-based-media-working-group/documentation.

National Archives. “Web Archiving Guidance,” 2011. https://cdn.nationalarchives.gov.uk/documents/information-management/web-archiving-guidance.pdf.

Obermann, Arnaud. Interview with Arnaud Obermann by Tom Ensom. Audio Recording, March 3, 2021.

Oleksik, Peter. “Email to the Authors: Book Project and a Question about 3D Printed Art.,” February 19, 2021.

Oleksik, Peter, and Megan Randall. “(Material Transfers & Translations) Tauba Auerbach’s Altar/Engine: A Case Study in Reconceptualizing Materiality.” Presented at the AIC’s 46th Annual Meeting, Houston, Texas, USA, June 1, 2018. https://aics46thannualmeeting2018.sched.com/event/Cz5w/material-transfers-translations-tauba-auerbachs-altarengine-a-case-study-in-reconceptualizing-materiality.

Open Source Initiative. “The MIT License.” Open Source Initiative. Accessed December 1, 2021. https://opensource.org/licenses/MIT.

Oracle Corporation. *VBoxManage (Software)*, 2004. https://www.virtualbox.org/manual/ch08.html.

———. *VirtualBox (Software)*, 2007. https://www.virtualbox.org/.

Paul, Christiane. “Email to the Authors: Time-Based Media Art Conservation: Source Code Analysis for Software-Based Artworks,” February 19, 2021.

Phillips, Joanna, Emma Dickson, Deena Engel, and Jonathan Farbowitz. “Restoring Brandon, Shu Lea Cheang’s Early Web Artwork.” *Guggenheim Blog* (blog), May 16, 2017. https://www.guggenheim.org/blogs/checklist/restoring-brandon-shu-lea-cheangs-early-web-artwork.

Phillips, Joanna, Deena Engel, Jonathan Farbowitz, and Karl Toby Rosenberg. “The Guggenheim Restores John F. Simon Jr.’s Early Web Artwork ‘Unfolding Object.’” *Guggenheim Blog* (blog), November 19, 2018. https://www.guggenheim.org/blogs/checklist/the-guggenheim-restores-john-f-simon-jr-early-web-artwork-unfolding-object.

Post, Colin. “Preservation Practices of New Media Artists: Challenges, Strategies, and Attitudes in the Personal Management of Artworks.” *Journal of Documentation* 73, no. 4 (July 10, 2017): 716–32. https://doi.org/10.1108/JD-09-2016-0116.

Puckette, Miller, and Varoius. *Pure Data (Software)*, 1996. http://puredata.info.

QEMU Project Developers. *QEMU (Software)*, 2010. https://www.qemu.org/.

———. *Qemu-Img (Software)*, 2021. https://www.qemu.org/docs/master/tools/qemu-img.html.

Reas, Casey. “Reas / Studio - Home (Website),” 2016. https://github.com/REAS/studio/wiki.

Rechert, Klaus, Patricia Falcão, and Tom Ensom. “Introduction to an Emulation-Based Preservation Strategy for Software-Based Artworks,” December 2016. http://www.tate.org.uk/research/publications/emulation-based-preservation-strategy-for-software-based-artworks.

Recode Project. “ReCode Project.” Accessed October 4, 2021. http://recodeproject.com/.

Rhizome. “Conifer (Software),” 2022. https://conifer.rhizome.org/.

Roeck, Claudia. “Conservation Case Study: TraceNoizer by LAN.” Accessed December 1, 2021. https://www.hek.ch/en/collection/research-and-restauration/conservation-case-study-tracenoizer-by-lan/.

———. “Email to the Authors: Time-Based Media Art Conservation: Source Code Analysis for Software-Based Artworks,” February 17, 2021.

———. “Emulating Horizons (2008) by Geert Mul: The Challenges of Intensive Graphics Rendering – Electronic Media Review.” *The Electronic Media Review* Volume Five: 2017-2018 (2018). https://resources.culturalheritage.org/emg-review/volume-5-2017-2018/roeck/.

———. Interview with Claudia Röck by Tom Ensom and Patricia Falcão, July 24, 2020.

Roeck, Claudia, Rafael Gieschke, Klaus Rechert, and Julia Noordegraaf. “Preservation Strategies for an Internet-Based Artwork Yesterday, Today and Tomorrow,” 2019. https://doi.org/10.17605/OSF.IO/GF2U9.

Roeck, Claudia, Julia Noordegraaf, and Klaus Rechert. “207.4 Evaluation of Preservation Strategies for an Interactive, Software-Based Artwork with Complex Behavior Using the Case Study Horizons (2008) by Geert Mul.,” 2019. https://doi.org/10.17605/OSF.IO/2VPFT.

Rösler, Wolfram. “The Hello World Collection,” 2021. http://helloworldcollection.de.

Rossenova, Lozana. “ArtBase Archive—Context and History: Discovery Phase and User Research 2017–2019.” Rhizome, 2020a. https://sites.rhizome.org/artbase-re-design/docs/1_Report_ARTBASE-HISTORY_2020.pdf.

———. “Artbase Redesign Data Models: Design Exploration and Specification Phases 2018–2019,” 2020b. https://sites.rhizome.org/artbase-re-design/data-models.html.

Rothenberg, Jeff. “Avoiding Technological Quicksand: Finding a Viable Technical Foundation for Digital Preservation: A Report to the Council on Library and Information Resources.” Washington, DC: Council on Library and Information Resources, 1999. https://clir.wordpress.clir.org/wp-content/uploads/sites/6/pub77.pdf.

———. “Ensuring the Longevity of Digital Documents.” *Scientific American* 272, no. 1 (1995): 42–47. https://www.jstor.org/stable/24980135.

Russel, Roslyn, and Kylie Winkworth. “Significance 2.0, a Guide to Assessing the Significance of Collections.” Adelaide: Collections Council of Australia, 2009. https://www.arts.gov.au/sites/default/files/significance-2.0.pdf?acsf_files_redirect.

Sherring, Asti, Mar Cruz, and Nicole Tse. "Exploring *the Outlands* : A Case-Study on the Conservation Installation and Artist Interview of David Haines' and Joyce Hinterding's Time-Based Art Installation." *AICCM Bulletin* 42, no. 1 (January 2, 2021): 13–25. https://doi.org/10.1080/10344233.2021.1982540.

Smithsonian. "Smithsonian TBMA Symposia." Smithsonian Institution, 2014. https://www.si.edu/tbma/smithsonian-tbma-symposia.

Software Preservation Network (SPN). "EaaSI Emulation-as-a-Service Infrastructure." Software Preservation Network. Accessed June 11, 2021. https://www.softwarepreservationnetwork.org/emulation-as-a-service-infrastructure/.

———. "SPN Software Preservation Network (Website)," 2022. https://www.softwarepreservationnetwork.org/.

Stricot, Morgane, and Matthieu Vlaminck. Interview with Morgane Stricot and Matthieu Vlaminck by Tom Ensom. Audio Recording, February 15, 2021.

Suchodoletz, Dirk von, Peter Bright, Remco Verdegem, Jasper Schroder, Paul Wheatley, and Jeffrey van der Hoeven. "Planets Project: Report Describing Results of Case Studies," May 30, 2008. https://web.archive.org/web/20201230191241/https://www.planets-project.eu/docs/reports/Planets_PA5-D3_CaseStudies-final.pdf.

Suchodoletz, Dirk von, Klaus Rechert, and Isgandar Valizada. "Towards Emulation-as-a-Service: Cloud Services for Versatile Digital Object Access." *International Journal of Digital Curation* 8, no. 1 (June 14, 2013): 131–42. https://doi.org/10.2218/ijdc.v8i1.250.

Swalwell, Melanie, and Denise de Vries. "Collecting and Conserving Code: Challenges and Strategies." *Journal of Media Arts Culture* 10, no. 2 (2013). https://www.academia.edu/3831859/Collecting_and_Conserving_Code_Challenges_and_strategies.

W3Schools. "HTML Element Reference." In *W3 Schools*. Accessed November 29, 2021. https://www.w3schools.com/TAGS/default.ASP.

Walker, Sarah. *PCem (Software)*, 2007. https://pcem-emulator.co.uk/.

Waller, Robert. "Conservation Risk Assessment: A Strategy for Managing Resources for Preventive Conservation." *Studies in Conservation* 39, no. sup2 (January 1, 1994): 12–16. https://doi.org/10.1179/sic.1994.39.Supplement-2.12.

Webrecorder (Software). Rhizome. Accessed February 7, 2022. https://webrecorder.net/about.

Wget (Software). GNU. Accessed February 7, 2022. https://www.gnu.org/software/wget/.

Wharton, Glenn. "Email to the Authors: Time-Based Media Art Conservation: Source Code Analysis for Software-Based Artworks," February 1, 2021.

Wharton, Glenn, and Deena Engel. "Museum And University Collaboration In Media Conservation Research." *The Electronic Media Review* Volume Three: 2013-2014 (2015): 8. https://resources.culturalheritage.org/emg-review/volume-three-2013-2014/wharton/.

Wheatley, Paul. "Digital Preservation and BBC Domesday." In *Electronic Media Group Annual Meeting of the American Institute for Conservation of Historic and Artistic Works*, 1–9. Citeseer, 2004.