Preserving Virtual Reality Artworks: White Paper

Tom Ensom & Jack McConchie
Time-based Media Conservation, Tate

Version: March 2020 Draft

Note: This is a pre-publication draft document and may contain some incomplete content, including missing publication, figure, table and section references (denoted with the text #*REF*).

Scope and Structure

This paper seeks to clearly introduce immersive media technologies (with a focus on virtual reality) and examine the challenges artists and the cultural heritage sector face in achieving their long-term preservation. The paper is divided into five sections, the first three of which focus on describing particular aspects of VR technology and identifying preservation risks associated with them: Section 1 describes the hardware and software constituents of VR systems; Section 2 describes 360 video, including its production and display; Section 3 describes real-time 3D VR software, including its production and display. In the final two sections, we recap the primary preservation risks, and consider how we might effectively respond to them: Section 4 discusses the preservation risks faced in collecting and caring for VR artworks and considers the suitability of existing preservation strategies for solving these; Section 5 offers a set of recommendations for the preservation of VR artworks, including suggestions for further research and development work.

Acknowledgements

This research was supported in part by Lumen Art Projects Ltd.

Thank you to the artists we interviewed during this research for their insight into their working practice: Dom Biddulph, Lawrence Lek, Michael Takeo Magruder, Winslow Porter, Jakob Kudsk Steensen, James Stringer, Milica Zec.

Thank you to Dragan Espenschied and Claudia Roek, with whom we organised the iPRES 2019 VR hackathon in Amsterdam, and to all those who participated in the event.

Thank you to the members of the Preserving Immersive Media Group for their insight and feedback on our work during the course of this project, and for all their comments on the first draft of this manuscript.

Contents

1. Introduction

Tate is a major arts institution that houses the United Kingdom's national collection of British, international and contemporary art. Tate's Conservation department works to ensure that this collection is appropriately cared for and remains displayable in the long-term. As part of the department's research programme, Tate keeps abreast of new technologies and their use in contemporary art, to ensure preparedness as the collection grows. This has led to a new strand of research on the preservation of artworks using *immersive media*, which we use to describe XR technologies (including virtual reality (VR), augmented reality (AR) and mixed reality (MR) technologies), which have been designed to immerse a user in a virtual space or combine virtual and real spaces.

Other research at Tate precedes this project. Firstly, a recent research focus on the conservation of software-based art (Falcão et al. 2010; Laurenson 2016; Rechert et al. 2016, Ensom 2019), a relatively new challenge for museums, had led into the development of novel workflows as part of Tate's Software-based Art Preservation Project. Recent acquisition of artworks using real-time 3D technologies (which are closely associated with VR) has presented further opportunities to refine these, including John Gerrard's *Sow Farm, near Libbey, Oklahoma 2009* (2009) and Ian Cheng's *Something Thinking of You* (2015). Research has also been carried out on the suitability of virtual reality (VR) technologies to document complex physical installations of time-based media artworks (McConchie 2018). Beyond work at Tate, we also build on earlier exploratory research around the preservation of VR (Campbell 2017; Cranmer 2017).

As a starting point for this research, we examined the current state of the art in VR technologies and identified the key components of a generic VR system at the level of hardware and software. We then considered the factors placing these components at risk of loss and how we might respond to these risks. We worked under the assumption that existing frameworks for Time-based Media Conservation in place at Tate and other institutions are at their core fit for purpose, requiring adaptation rather than reinvention. Consequently, our approach assumes that preservation techniques such as software migration and hardware emulation might be applied, while ensuring that the work-defining characteristics of the artwork are maintained as far as possible. The new knowledge required to do this effectively is to identify the characteristics that define VR artworks and how those might link to the specific technologies employed. In an industry that develops new technology at a rapid pace, the interests of a collecting institution can be seen to quickly diverge from technological change driven by innovation and profit. With this in mind, we devote the first three sections of this paper to examining specific component groups of VR artworks: VR systems, real-time 3D software and 360 video. For each we attempt to identify points of variability and risks and potential strategies for their long-term preservation. We also offer a set of pragmatic and propose sector-level goals for future research and advocacy.

2. VR Systems

2.1. VR System Hardware

VR systems involve a system of interlinked hardware components, which will typically include a head-mounted display (HMD), tracking system and controller(s), all of which are connected to a host computer. The relationships between these components are described in Figure #REF. These systems are ultimately centered on the user through the tracking of their movements and controller inputs, data from which controls interaction within virtual space and thus the generation of the frames sent to the HMD.

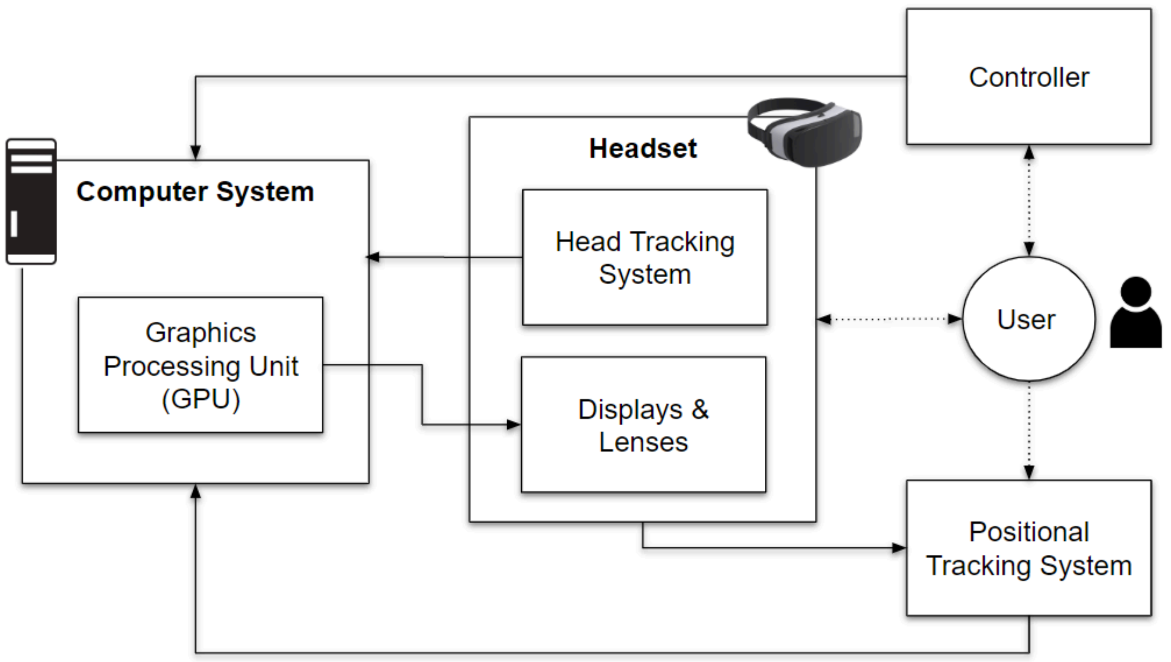


Figure #REF. Diagram of a typical VR hardware system.

In the majority of cases, all of these components must be present in order for an VR artwork to be displayed, which creates immediate preservation risks through the potential for failure of the hardware used. Much as for other specialised hardware, as time passes obsolete components which stop functioning are likely to become impossible to replace with an equivalent device. The hardware systems sold and supported by VR companies currently move through rapid development and obsolescence cycles. For example, Oculus launched the consumer Rift in March 2016 and replaced it with the Rift S in March 2019, while the Microsoft HoloLens, released in March 2016, was superseded by the HoloLens 2 in February 2019.

We may stockpile devices to prevent immediate loss but this offers only a short term solution, particularly when devices such as HMDs and controllers must be physically interacted with by those experiencing a work which will lower their lifespan. Furthermore, the model of stockpiling and specialist repair that museums have employed to maintain access to CRT television technology has worked thus far because these technologies were ubiquitous during their era of production and sale. The same could not be said for VR hardware, which despite growing commercial interest has remained of relatively niche interest. Specialist networks could help us maintain the hardware we are able to acquire but the closed source and proprietary nature of this hardware remains an impediment to this.

If hardware obsolescence is inevitable within this fast moving industry, we need to determine whether we can manage the impact of changes to the hardware components used in the realisation of VR artworks. To do so we need to understand two issues. The first is which characteristics of the hardware impact the experience of that artwork, so that we can seek to preserve them independently of the original hardware itself, if appropriate. This is addressed in the next part of this section as the specific parts of an VR system are introduced in more detail, including consideration of the characteristics which may vary between models within the current generation of hardware. Secondly, we need to understand how the VR software connects with the hardware and how this link can be maintained if the specific hardware components change. Here we must consider the various linkages between hardware and software components, perhaps the most significant of which is the potential for incompatibility between VR application software (i.e. the software created by an artist) and a specific VR hardware system accessed via manufacturer-specific runtime and driver software. This issue is discussed further in the VR System Software section below.

Head-Mounted Displays

A head-mounted display (HMD) is a display device which is attached directly to a user's head. These devices contain either one or two screens, positioned in order to display a monoscopic or stereoscopic image at close proximity to the human eye. This close proximity provides a wide field of view for the user in order to increase immersion, while stereoscopy allows the simulation of realistic depth perception. HMDs can be tethered (i.e. connected to a PC via physical cabling) or untethered (i.e. either standalone or connected to a PC wirelessly). At the time of writing, the latter is typically achieved by inserting a mobile device into an HMD (e.g. the Samsung Gear VR) or using mobile technology integrated into the HMD itself (e.g. the Oculus Quest).

We have identified the following characteristics as creating variability among HMDs:

- Screen panels: Variation in the type (LCD/OLED)[1], colour reproduction, pixel density, refresh rate, aspect ratio and latency of embedded panels.
- FOV: The extent to which the LCD panels completely cover the user's field of view.
- Lenses: In order to achieve clear visibility of the panels at close range across at such a wide field of view, thick Fresnel lenses are placed between the eye and the panels. The image distortion they introduce must be corrected for in the frames sent to the HMD (see Section #REF) using a hardware-specific algorithm (see VR Runtimes #REF) which can result in variation in the quality of the image across the field of view.
- User experience: The experience of wearing a HMD can vary considerably based on its physical construction, such as eye piece cushioning, strap design and overall weight.

Tracking Systems

Tracking systems are used to capture the users rotational and positional movement within physical space, so that this can be translated into movement in virtual space. There are two dominant technical approaches to implementing tracking systems known as *inside-out* and *outside-in* tracking respectively. Inside-out tracking uses sensors placed on the HMD. Outside-in tracking uses sensors mounted in the external environment. Both approaches may utilise markers placed in the environment or on the HMD to improve tracking.

We have identified the following characteristics as creating variability among tracking systems:

[1] https://realnewworld.com/vr-hmd-display-technology/

- Area: Potential size of tracked area, including maximum, minimum and shape limitations,
- Resolution: Density of tracking data the system can capture and transmit for software processing.
- Latency: The time taken for tracking data to be processed and translated into movement in virtual space.
- Occlusion: Ability of the system to tolerate occlusion (i.e. blocking of a tracking device or marker).
- Hand/finger tracking: Support for hand or finger tracking, which can be used as a means of interaction with the virtual environment.
- Eye tracking: Support for eye tracking, which can be used in techniques like foveated rendering or as a means of interaction with the virtual environment.
- Virtual boundary representation: The way in which the software represents the boundary limits of the interactive area to the user (e.g. Oculus Guardian).

Controllers

Controllers are physical devices which allow a user to interact with the virtual environment. In some cases an VR system may use a traditional hand-held video game controller. VR hardware manufacturers have also experimented with various forms of custom controller, all of which prioritise freedom of arm movement in comparison to traditional video game controllers which require the two hands of the user to be held in relative position. The Oculus Rift headset was sold with trigger controllers, while the Vive came with wand-type controllers. In some cases these controllers may be visibly represented inside the virtual environment to create a continuity between physical sensation and the perceived virtual space.

We have identified the following characteristics as creating variability among controllers:

- Button/stick/pad orientation: The layout of the interactable elements of the device create a particular user experience (e.g. the design of Oculus Rift trigger controllers lends itself to firing weapons).
- Haptics: Physical feedback can be created to the users controller inputs with the aim of reinforcing immersion.
- Virtual representation: A controller and/or player hands can be represented inside the virtual environment, again with the aim of reinforcing immersion.

Computers

Computer systems orchestrate the execution of the software and its interaction with the VR hardware. While the high performance demanded by VR applications has tended to require powerful desktop PCs, mobile technology has also been extensively experimented with, particularly for use with relatively lightweight 360 video. This has resulted in all-in-one headsets like the Oculus Go (which relies on standard mobile technology built into the headset itself[2]) and the GearVR, which is a HMD designed for pairing with a Samsung mobile phone. The key hardware components in a desktop computer suitable for VR applications are described in Table #REF.

Name	Description
Central Processing Unit (CPU)	The processor executes native code when an application is executed. Native code is compiled for a particular instruction set architecture (ISA) e.g. the x86-64 ISA in desktop computing, or ARM ISAs in mobile computing.
RAM	Fast, volatile memory to which programs are loaded when they are executed.
GPU	A specialised piece of hardware for rendering 3D graphics. While typically taking the form of standalone expansion cards, GPU functionality can also be integrated into a CPU, although this tends to be unsuitable for VR.
Storage	Devices containing a software environment, usually consisting of an operating system, software programs and user data. SSDs provide faster speeds so are often favoured over traditional magnetic HDDs for high performance VR applications.
Interfaces	VR systems use a large number of peripheral devices which must be connected to the host machine (typically at least 2x USB 3.0 for tracking and sensor data and 1x HDMI for sending frames to the HMD.

Table #REF. Key components of a VR-ready desktop computer.

As a result of the very large array of individual components available with which to construct computer systems, understanding the variance they introduce is beyond the scope of our research. However, a close examination of case studies during our research has highlighted two key points. The first is that the primary purpose of hardware selection when creating a system to support an VR application is performance. Each component can be critical in lowering latency and increasing the speed with which frames are generated. The second is that the GPU is of primary importance within this constellation of components, given its critical role in the processing of shaders and the creation of the frames sent to the HMD. Different GPU models, in combination with the specific driver versions installed, support different rendering capabilities. For example, to use features of Shader Model 5.0, you would need to use a GPU and driver combination supporting Shader Model 5.0. Emulated GPU devices tend to have limited functionality in comparison to their hardware equivalents, a fact which makes emulating 3D applications challenging, particularly those with high performance requirements such as VR applications. We discuss this issue, a possible short term solutions, in Section 5 #REF.

2.2. VR System Software

VR systems depend on a stack of software components, a generic version of which is illustrated in Figure #REF below. The software system centers on the VR application itself,

[2] https://mindtribe.com/2018/09/oculus-go-teardown/

the execution of which engages these other layers. VR applications are not discussed in this section in detail but are covered in detail in Sections 3 (for Real-Time 3D) and 4 (for 360 Video). This section instead focuses on understanding the variance introduced by the other software layers on which the application depends.

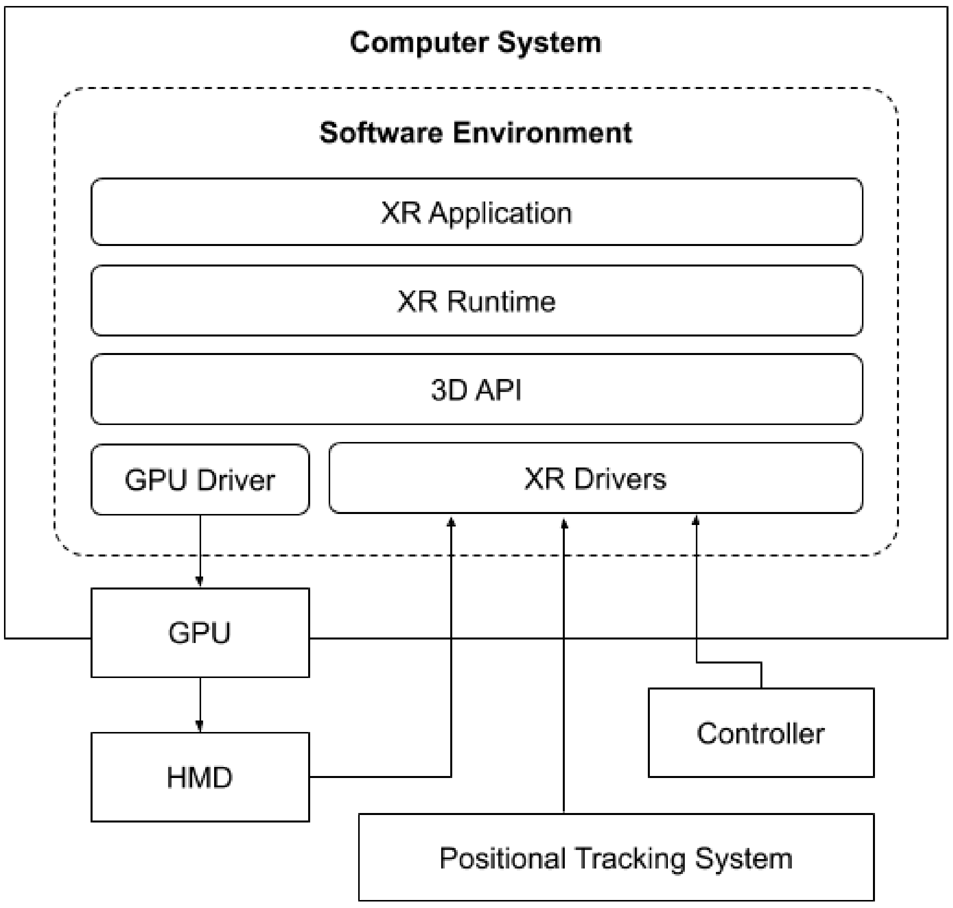


Figure #REF. Diagram of components and communication pathways in a generic VR software environment.

An VR software application may have varying degrees of dependency on these environment components: hard dependencies are those which cause the software to stop functioning, while soft dependencies are those which impact significant characteristics. Dependencies arise through design decisions during the development process, usually because of the choice to target a specific set of hardware. The generic core components of VR suitable software environments are summarised in Table #REF below and discussed further in the following sections.

Name	Description	Examples
Operating System	Supports a computer's basic functions, including managing interactions between software and hardware.	Windows, Mac OS, Linux

3D API	Abstraction layer for software to access the features of graphics hardware. Typically closely linked to OS e.g. DirectX is a core component of Windows.	Vulkan, DirectX, OpenGL, Metal
VR Runtime	Provides software with access to VR hardware and implements features which affect rendering like lens distortion.	Oculus, OpenVR (Vive/SteamVR), Windows Mixed Reality
Hardware Drivers	Specialised software controlling a connected device. For consumer oriented VR products, hardware drivers are typically bundled with and managed by the VR runtime.	GPU/display, headset, controllers, positional tracking system
Additional Libraries	Any additional necessary libraries not found in the operating system or VR runtime.	.NET/Mono runtimes, MSVC++ runtimes

Table #REF. Description of the core components of a generic VR system with real-world examples of each technology.

In general, a few important considerations have become clear from our research. The first is that contemporary software environments are highly volatile due to the nature of always-on internet access and automatic updates. Several of the artists we spoke to commented that automatic updates to software components make maintaining access to specific versions of their software challenging. In some cases, they have resorted to creating dedicated offline systems to ensure they remain static snapshots of the software environment. It is clear that if we hope to preserve and manage such an environment, particularly if we hope to effectively apply strategies such as emulation, we need to ensure we archive snapshots of working software environments. We recommend that the storage volumes containing software environments for artist-verified systems be captured using disk imaging for preservation purposes - a topic we return to in Section 5 #REF.

VR Runtimes

VR runtimes are software packages which orchestrate communication between application software and VR hardware. VR runtimes are often closely tied to the manufacturer of VR hardware. A brief summary of these which is given in Table #REF below.

Name	**Description**	**Target Platforms**
SteamVR	Originally created for Valve’s own Vive VR hardware, it has since expanded to include support for other VR	Windows, Linux, MacOS (beta only)

	systems. Implements Valve's own OpenVR[3] specification, but is not itself open source[4].	
Oculus	Oculus's runtime for their Rift VR systems. Note: Oculus Go has the mobile version of the runtime installed on the HMDs embedded hardware.	Windows, MacOS, Linux
Open Source Virtual Reality (OSVR)	An open-source VR runtime intending to add support for all major VR hardware. It's future is uncertain, as commits to their GitHub repository have been infrequent since 2017. The main contributor, Ryan A. Pavlik, is now working on the OpenXR specification.	Windows, Android, Linux (partial support), Mac OS X
Daydream / Cardboard	Google's mobile-only platform for VR. Uses 'Google VR Services' on supported Android versions and phones.	Android
GearVR	A mobile-focused VR headset developed by Samsung, which requires the use of a compatible Samsung Galaxy mobile phone.	Android
Windows Mixed Reality	Supports a variety of Windows Mixed Reality headsets and the HoloLens headsets. Has OpenXR support.	Windows
PSVR	A VR system for Sony's Playstation 4. Requires a licence to develop for — no public SDK/engine/tools.	Playstation 4
ARKit	Apple's augmented reality (AR) platform. The runtime is integrated into iOS 11 onward and supported hardware.	iOS
ARCore	Google's mobile-only augmented reality (AR) platform. Uses 'Google Play Services for AR' on supported Android versions and phones.	Android
Monado	A Linux runtime implementing the OpenXR specification under development by Collabora.	Linux

Table #REF. Description of VR runtimes for which development is still active.

From a user perspective, this typically involves using an executable installer which carries out the downloading of up-to-date versions and their installation and configuration behind the scenes. This may actually involve a myriad of software programs, including drivers, libraries and management GUIs. They can also be closely integrated with the target operating system, particularly for mobile VR. From a preservation perspective, these factors, in addition to the frequency of updates, makes collecting and archiving specific runtime versions challenging. The only way we have identified to ensure all elements of a runtime are captured is to carry out imaging of the entire software environment.

[3] More info about OpenVR here: https://skarredghost.com/2018/03/15/introduction-to-openvr-101-series-what-is-openvr-and-how-to-get-started-with-its-apis/

[4] https://github.com/ValveSoftware/openvr/issues/154

The importance of VR runtimes is largely functional, meaning there is little variance introduced by the choice of runtime. The key features supported by a runtime are rotational tracking, positional tracking and lens distortion correction. In addition each runtime has a visual representation of a boundary at the limits of safe physical space, as set up by the user. As HMDs currently use thick lenses to maximise the field of view at close viewing distance to the panels, frames sent to the headset must be distorted to compensate for distortion these lenses introduce. Techniques can be used to improve the quality of the distorted image. Supersampling (i.e. rendering at a higher resolution and downscaling) can be used to further compensate for the loss of pixel data in the outer edges of the rendered frame as a result of the distortion correction process. Lens matched shading increases resolution consistency by rendering an image which more closely matches the distorted images sent to the HMD[5]. Lens distortion processes are handled by the proprietary VR runtime and the distortion algorithm used may not be made public. This creates a strong dependency on using an VR runtime which understands the distortion algorithm employed by a particular HMD. Techniques have been developed to derive distortion algorithms manually using photography[6].

Additional proprietary techniques can be implemented by the VR runtime to improve performance. Reprojection techniques (such as timewarp and spacewarp) are used by the current generation of VR runtimes to artificially increase framerate when a host system is unable to generate frames at the rate required to avoid discomfort in VR[7]. These techniques generate extra frames by distorting the previously rendered frame based on the movement of the user and the scene, offering a very efficient way to artificially raise framerates. While we feel that these techniques are all historically interesting, we do not consider maintaining their presence a priority for preserving VR artworks, as they are simple tools used to achieve higher framerates, are generally not apparent to the user and could be replaced by other software providing similar functionality or may simply become unnecessary due to advances in VR performance.

VR runtimes primarily create preservation risks through their close links with physical VR hardware and a requirement for application-level support. Without the installation of an appropriate runtime on the host computer system, VR hardware may function in a limited way (e.g. without appropriate lens distortion correction) or not at all. The VR application must also have support for the runtime built-in. Adding support for a specific VR runtime to an VR application requires adding support for the appropriate API at the application level. In Unity and Unreal Engine, this typically involves installing certain plugins or SDKs. From the perspective of an artist who wants to utilise a specific set of VR hardware, the use of hardware-specific runtimes means that they must build this support into their application during development or it will not be available in the packaged binaries.

Under the model described above, migration to new hardware in the future will be challenging as it is likely to require access to a new runtime or runtime version. The development of open runtime standards may help alleviate this problem. There is currently

[5] https://developer.nvidia.com/vrworks/graphics/lensmatchedshading
[6] https://github.com/OpenHMD/OpenHMD/wiki/Universal-Distortion-Shader
[7] https://uploadvr.com/reprojection-explained/

an open-specification runtime standard under development by the Khronos Group called OpenXR[8], which aims to standardise the connections between VR applications and VR runtimes, and VR runtimes and VR hardware (see Figure #REF). It is interesting to note that these are considered distinct goals, thus allowing manufacturer-specific VR runtimes a continued place within the VR software ecosystem, providing they can speak the language of OpenXR.

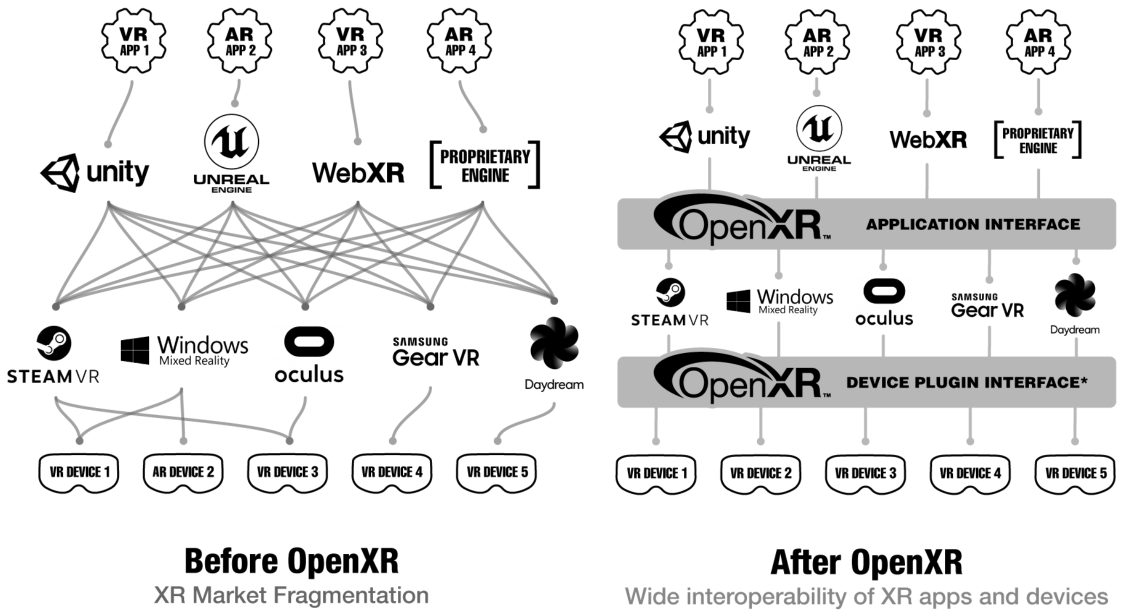


Figure #REF. The OpenXR standard is being developed by the Khronos Group, a tech industry-wide consortium, with the hope that it may solve the problem of VR market fragmentation.

Adoption of such a standard may help a preservation use case in several ways. For example, we could run an old VR application built with OpenXR support on contemporary VR hardware, providing OpenXR is available for that platform and backwards compatibility is maintained. This is uncertain because we don't know if Khronos see maintaining backwards compatibility as a priority, and we will remain dependent on implementations of that standard being available for a particular platform. For proprietary platforms, these may remain subject to the whims of their maintainers, as exemplified by Apple's recent decision to deprecate OpenGL on MacOS in favour of their own Metal API[9]. To maximise its value for preservation then, we would need OpenXR to maintain backwards compatibility with earlier versions of the standard and for it to be widely adopted and supported in the long-term by the various industry groups which produce VR technology. 3D engines and VR hardware are both unlikely to be created by artists themselves, therefore control of implementation of OpenXR lies with the companies that have the resources to develop and maintain these technologies. OpenXR is currently in its early stages and only a draft specification has been released with two public runtime implementations at time of writing: Monado[10] for Linux (which is open

[8] https://www.khronos.org/openxr
[9] https://arstechnica.com/gadgets/2018/06/the-end-of-opengl-support-other-updates-apple-didnt-share-at-the-keynote/
[10] https://monado.freedesktop.org/

source) and Windows Mixed Reality. On the engine side, support has been added to Unreal Engine 4.23 in the form of an OpenXR plugin.

Regardless of potential stumbling blocks, using an OpenXR implementation will have significant advantages over existing runtimes in that they implement an *open* standard. A specification of the standard is publicly available, which allows software to be written in compliance with the standard that can communicate with any other software that claims compliance. This means we have the possible fallback of developing compatibility software to map software written to support older versions of OpenXR with newer versions of the specification. Evidence of community efforts to alleviate compatibility issues across runtimes suggests that this is not only possible, but of interest to VR user communities. For example Revive[11] allows Oculus runtime exclusive applications to be experienced using a Vive, while OpenOVR[12] allows the inverse for SteamVR applications. Given ongoing interest in VR from gaming communities, we can be hopeful that such a solution might be developed by those interested in legacy games — work which could be effectively supported by cultural heritage institutions.

Operating System

Operating systems are responsible for the execution of an VR application as a computational process. They also provide part of the implementation of a 3D API (see dedicated section below), as well as a number of generic drivers (e.g. audio drivers, bluetooth drivers) required by VR systems. Analysing the impact of operating systems on VR systems in detail is beyond the scope of this paper due to their complexity and the scarcity of information for proprietary, closed-source operating systems like Windows and MacOS. In understanding the VR software stack it is useful to understand the 3D APIs they support, and whether they natively support VR hardware and thus the use of direct mode access to HMDs (see VR Device Drivers #REF section). This information is summarised for key VR operating systems in Table #REF below.

Operating System Family	3D API Support	Version Native VR Support Added
Windows 10	DirectX 9, DirectX 10, DirectX 11, Vulkan, OpenGL	Windows 10 Fall Creators Update (1709) 10.0.16299
MacOS 10	OpenGL (deprecated), Metal	macOS 10.13 (High Sierra)
Linux (Ubuntu)	OpenGL, Vulkan	X.org server 1.20, Linux kernel 4.15 and Mesa 18.2 (for SteamVR)

Table #REF. Operating systems currently supporting VR and their 3D API support.

3D API

[11] https://github.com/LibreVR/Revive
[12] https://gitlab.com/znixian/OpenOVR

3D Application Programming Interfaces (APIs) are specialised interfaces designed to make writing programs that use 3D graphics easier. They are typically implemented in part by the operating system and in part by the GPU driver software, both of which must be present for an application to be able to use that particular API to access GPU hardware. 3D APIs are a ubiquitous component of modern VR development (and real-time 3D development more generally). The application itself must be built to support a particular 3D API.

3D API	Operating System Support	Description
Direct3D	Windows 10	Proprietary API that has been part of Windows OS's since Windows 95 (as part of DirectX). Support on other platforms is limited to Wine's implementation for Linux.
Vulkan	Windows 10, Linux	Successor to OpenGL developed by the Khronos Group, this open standard is a low-level API designed to compete with Direct3D 12 and Metal.
OpenGL	Windows 10, MacOS, Linux	Open standard for 3D APIs implemented on many platforms (and on mobile through the OpenGL ES variant). Developed and maintained by Khronos Group.
OpenGL for Embedded Systems / OpenGL ES	Windows, Linux, Android	Open standard for 3D APIs on embedded devices and portables such as mobile phones. Developed and maintained by Khronos Group.
Metal	MacOS	Apple's low-level 3D API intended as a competitor to DirectX 12 and Vulkan.

Table #REF. List of contemporaneous 3D APIs

The current generation of 3D APIs are relatively homogenous[13], which is evidenced by the possibility of cross-platform build options in modern real-time 3D engines. In both Unreal Engine and Unity engine applications can be built to support any of the APIs listed in Table #REF. For Unity, the separation of the custom application content and the Unity player component means that an application can be executed using any of the APIs supported by the player component (this choice can be controlled using launch options).

As a result of their similarity, the 3D API used (when multiple options are available) is unlikely to introduce variability in rendering characteristics. The choice of API is more significant in terms of preservation due to the potential loss of API support from future graphics drivers and operating systems. There are indications that these kind of changes will happen in practice, a notable example being Apple's choice to deprecate the use of OpenGL on MacOS in favour of their own Metal API. The choice between open standards (such as OpenGL and Vulkan) and proprietary, platform specific APIs (such as Direct3D and Metal)

[13] This article has more detailed on the differences: https://alain.xyz/blog/comparison-of-modern-graphics-apis

may also be significant as we have a better chance of being able to keep those built on open standards accessible in the future, as writing compatibility software is made much easier.

VR Device Drivers

VR devices may use either generic OS specific drivers or additional third-party drivers. The exact driver used by a piece of hardware are not necessarily easy to identify as they are typically packaged with an VR runtime installer (e.g. SteamVR on Windows uses both .sys files managed by the system registry and so-called .dll 'minidrivers' which are loaded at runtime) and managed behind the scenes. As a result, while dynamic analysis tools can be used to help reveal these dependencies, but effectively archiving them remains difficult outside of capturing a full software environment image (see Section #REF).

Since the first wave of commercial VR HMDs, HMD drivers usually make use of *direct mode* features provided by the GPU driver and 3D API. This allows an application to treat an VR HMD as a dedicated and distinct display device, rather than as an additional display which would extend the desktop. *Extended mode*, which simply treated VR HMDs as additional monitors, was used prior to this but has fallen out of general use due to the amount of configuration required and a desire to improve user experience from hardware manufacturers. At time of writing, extended mode remains accessible in current versions of SteamVR but has been removed from the Oculus runtime.

VR drivers are largely functional and present little variance in the features they provide for similar HMDs. In some cases, alternative drivers can be used, for example the open-source OpenHMD[14] package, although often at the cost of the complete set of functionality offered by the manufacturer specific VR runtime. For example, OpenHMD currently supports only rotational tracking and, as it is designed to replace the runtime and driver set provided by a hardware manufacturer, cannot make use of runtime specific techniques like reprojection (see Section #REF). The OpenHMD wiki documentation on GitHub[15] is a useful resource for learning more about VR hardware drivers, which are usually proprietary and closed source projects.

GPU Drivers

GPU drivers allow the operating system and application to interact with specialised graphics processing hardware (or GPU devices). Beyond supporting 3D rendering capabilities, GPU drivers implement device-specific support for direct mode VR (for example, NVIDIA added support for Oculus direct mode in 355.83, HTC Vive direct mode in 361.75 and Windows Mixed Reality in 387). Drivers also form an essential part of the implementation of different 3D APIs. The GPU driver is also a critical component in ensuring high performance in VR applications. All of these factors are concerns for long-term preservation, as making changes to the rendering hardware may also require changes to the GPU driver, potentially resulting

[14] http://www.openhmd.net/
[15] https://github.com/OpenHMD/OpenHMD/wiki/Getting-Started

in loss of access to VR devices in direct mode and applications which use particular 3D APIs.

3. Real-Time 3D VR

Real-Time 3D (RT3D) VR artworks use software which instructs the host VR system as to how moving images are rendered in real-time. In order to understand how to preserve RT3D VR applications, we need to understand how they are created and how they function. In this section, we will introduce the production process for real-time 3D applications, focusing on the engine-based approach taken by the artists we interviewed. Given existing precedent for collecting source materials in software preservation (#REF Engel), we will consider potential value and the practical implications of collecting production materials associated with VR applications. In the next section, we will examine the compiled software or *builds* which are the primary output of the production process. In the final section, we will explore approaches to the creation of conservation documentation for real-time 3D VR applications.

3.1. Real-Time 3D VR Production Materials

The process of producing RT3D VR artworks typically involves bringing together an array of *assets* — various data types including 3D models, textures and audio — in a *3D engine* — a specialised piece of software which provides a production environment for creating real-time 3D environments which can then be exported as packaged software applications. These two components are discussed in more detail in the following sections.

Engines and Project Files

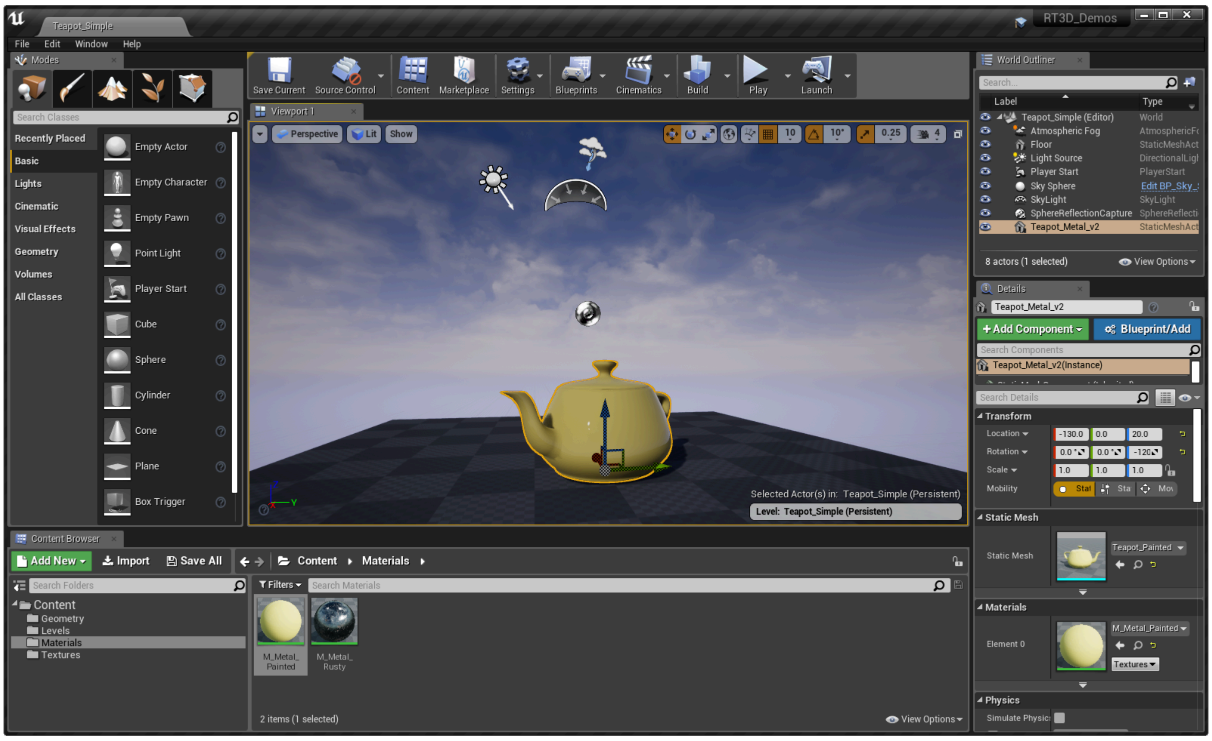
Real-time 3D engines are production environments for creating real-time 3D software. They integrate a broad range of functionality, typically centered around the manipulation of 3D scenes through a viewport. Components provided by a contemporary game engine typically include:

- Renderer: Generates animated 3D graphics in real-time from the assembled data sources (geometry, materials, lighting and particle systems etc.).
- Shader Management: Enables authoring of shaders (small rendering programs which run on the GPU) and translation[16] into 3D API specific languages such as HLSL (Direct3D), GLSL (OpenGL) and SPIR-V (Vulkan)
- Scripting: Allows authoring and compilation of dynamic or interactive behaviours using programming languages or equivalent interfaces.
- Physics engine: Simulates physical forces and the resulting interactions between geometry.
- Audio engine: Allows playback and manipulation of audio, including positional and spatial components.
- Cross-platform build support: Allows compilation of projects to binaries for different platforms, including various operating systems, 3D APIs and VR runtimes.

[16] Compilation or interpretation of shaders is handled by the GPU at runtime

- Asset management: System for the import, export and organisation of assets such as textures, 3D models and audio files.

Much like the VR systems themselves, the real-time 3D engines used in VR have found their primary market in the video game industry. While larger game studios might develop their own engine, many will licence a third-party engine such as Unity and Unreal Engine 4 (UE4) [17]. Third-party real-time 3D engines are useful because much of the functionality required when creating real-time 3D software is generic: there is little value to engaging in the lengthy process of implementing a 3D renderer or physics engine when an existing implementation can meet your requirements and allow focus on more creative elements of the production process. This is particularly appealing to artists, which may not be approaching the creation of VR content with a low-level understanding of 3D rendering or as programmers at all. The artists we spoke to worked exclusively with the third-party engines Unity and Unreal Engine 4.



Location -130.0 0.0 20.0
Rotation 0.0° 0.0° -120°
Scale 1.0 1.0 1.0

Figure #REF. A screenshot of a simple 3D scene in Unreal Engine 4.22.

Unity and Unreal Engine 4 are both free to use software packages which employ different licensing models to control how they are used. Unity uses a tiered subscription model which requires users with revenue or funding of over $100,000 over the past 12 months to buy a paid plan, which rises in cost in several further tiers based on revenues[18]. Unreal Engine 4 uses a licencing model which requires users to pay 5% royalties to Epic Games if their

[17] Limited data is available to make meaningful estimates as to how many, but this survey provides a useful baseline: https://www.reddit.com/r/gamedev/comments/8s20qp/i_researched_the_market_share_of_game_engines_on/
[18] https://store.unity.com/compare-plans

products earn over certain $3,000 per quarter[19]. Both engines have accessible source code repositories, although the code itself is not distributed under an open-source licence. A partial C# component of the Unity source code is publicly accessible on GitHub but does not allow modification or redistribution. Full source access and modification rights are only available with a higher tier paid subscription and an individually negotiated source access agreement. The full Unreal Engine 4 source code can be read and modified by anyone agreeing to the Unreal Engine EULA but not redistributed in this form.

Encouraging the preservation of the source projects of real-time 3D applications seems a logical approach to their long-term preservation, opening up options of migration and modification. The Unity and Unreal Engine 4 engine project formats that we have examined exist within single structured directories, which can be archived as-is to capture the hierarchy of assets, levels and other materials. However, they remain contingent on the appropriately versioned engine binaries for access. This presents a significant challenge on several fronts. Firstly, engines cannot be considered entirely stand-alone pieces of software and are typically downloaded and installed by installer applications which conceal details of the configuration. Once installed, the engines may have additional dependencies on third-party pieces of software in order to build for certain platforms (e.g. Google's Android SDK is required to create Android builds). Furthermore, there is a high frequency of updates made by their developers and there is a tendency to remove access to old engine binaries over time. For example, at the time of writing versions of Unreal Engine binaries prior to version 4.0.2 (released 28 March 2014) are no longer publicly accessible, while accessible Unity binaries extend back as far as version 3.4.0 (released 26 July 2011). A final limitation to note is that even if we archive a source project and engine binaries, without archiving complete engine source code (only possible Unreal Engine 4 at time of writing) we have only a partial representation of the software.

Scenes and Assets

Assets are the various data types which are imported into game engines and used in the construction of virtual *scenes* — alternatively and more game-centrically called levels or maps, or more technically understood as a *scenegraph*. Assets can include a huge array of file types including 3D models, raster images and audio. There may be a desire or need to preserve assets independently of any engine as this approach leaves additional options open for future preservation efforts at a lower level of granularity than the engine project files. The creation of assets involves various specialised workflows, a detailed analysis of which is beyond the scope of this paper. It is clear from our limited research that in many cases (for example, for audio or raster images) preservation pathways will converge with those for similar file types in other domains. In this sense, preserving game engine assets fans out into an overwhelmingly broad consideration of preservation issues across many different file formats and tools. We will focus our discussion here on an asset type which is particularly significant in real-time rendering (and relatively novel within preservation literature): the 3D model. In this section we will devote some time to discussing what

[19] https://www.unrealengine.com/en-US/faq

constitutes an engine-ready 3D model and a consideration of the file format stabilisation options available at the current time.

At its most basic, a 3D model is a set of point coordinates (known as *vertices*[20]) which describe the structure of the surface of a 3D object. Each vertex can have properties, such as a *normal*, which describes the direction it is visible from, and also incorporate UV map coordinates which describe how textures are mapped over the models surface. Accompanying the geometry is information regarding *materials*, which describe the characteristics of a 3D model's surface when it is lit in an engine. In the early days of 3D rendering this might have simply been a square raster image tiled over the surface of the model, but in modern 3D rendering can include many layers of texture *maps* which describe different properties of the material so that physically accurate results can be achieved when the surface is lit. This approach to creating materials is known as *physically-based rendering* (PBR) and typically uses texture maps describing color (also known as a diffuse or albedo map), surface detail (known as normal maps), shadowing (ambient occlusion maps), roughness (or in some workflows glossiness), metallicness and specularity.

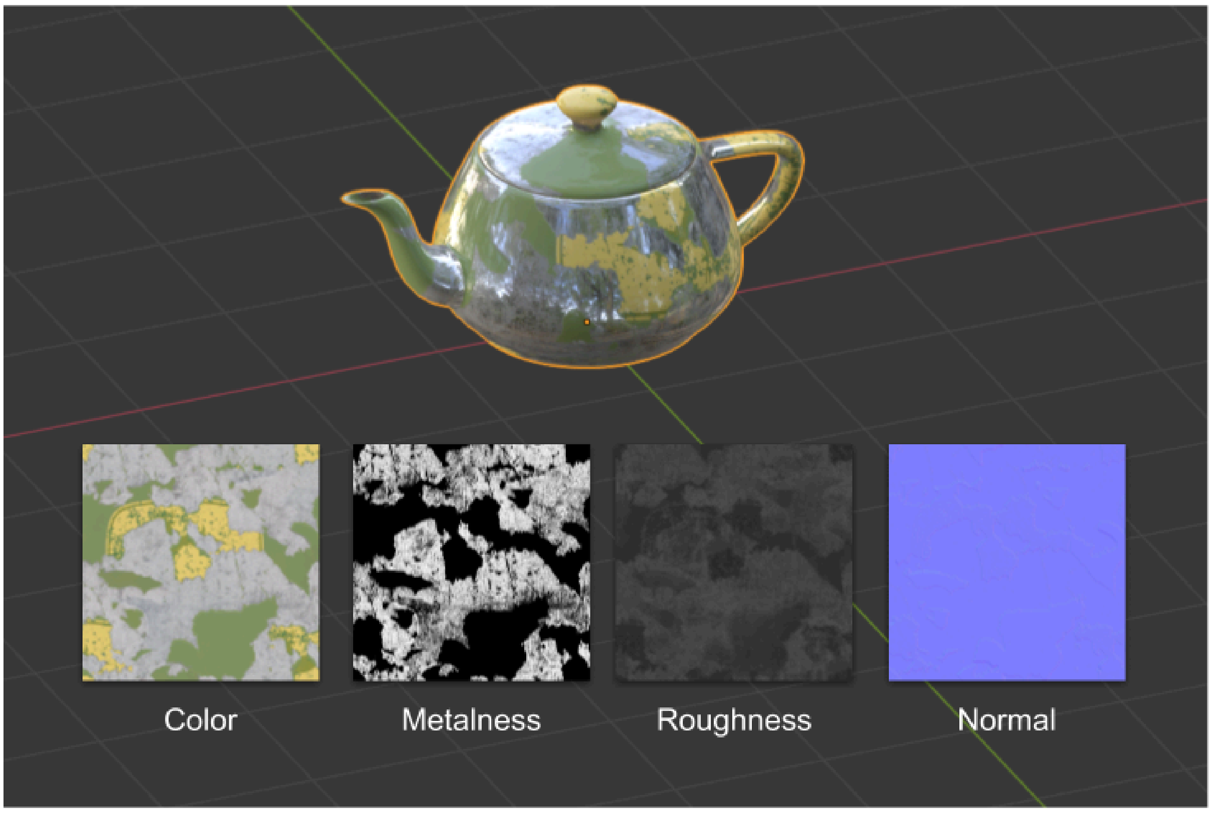


Figure #REF. These four texture maps have been used to apply a PBR material to a 3D model in Blender 2.8. Material textures downloaded from freepbr.com.

In combination, these maps allow the renderer to infer how light would bounce off that surface, and thus determine how to colour the pixels that represent that surface. Materials might in some cases be treated as a part of a 3D model or as a distinct concern, depending on the file format and workflow adopted.

[20] The singular form of vertices is *vertex*

There are an array of tools available to create engine-ready 3D models, with popular tools that we encountered including 3D Studio Max, Blender (the primary free and open source option in this domain), Houdini (which is specialised in procedural animation), Maya and ZBrush (which is specialised in sculpting). In addition to their own native formats, these tools are capable of importing and exporting 3D models in a variety of file formats in order to accommodate varied production workflows. Additional tools may be used in texturing workflows beyond widely used raster graphics editing tools like Photoshop and GIMP, such as the Substance Painter tools which are used for the creation of PBR materials. A 3D model suitable for usage is real-time 3D applications, particularly in an VR context, may be heavily optimised in order to improve performance. This typically involves reducing the complexity of geometry and lowering the resolution of texture maps.

Given the relative novelty of 3D model file formats as a digital preservation research topic, we are not yet at a stage where community consensus of preservation suitable formats has been reached, and a detailed comparison in the context of our research was beyond the scope of this project. The best resource at time of writing is the Library of Congress' 'Sustainability of Digital Formats' website[21], which includes a number of in-depth 3D file formats as part of an ongoing programme of reviews. Based on our preliminary research, we suspect that arriving at a recommendation for RT3D artwork asset preservation may not be possible in the current landscape. Fragmentation in requirements from different user groups means that 3D model formats are heterogenous and domain specific. The more generic formats with support across many applications — of which FBX, a proprietary format owned by Autodesk, is favoured in real-time 3D production — are so-called interchange formats, which often contain only partial representations of the content created in the authoring tool and require manual reconfiguration when imported into another tool. While there have been efforts to introduce open standards for 3D model file formats (for example, the COLLADA, U3D and X3D formats), they are hindered by the fast moving nature of the 3D graphics industry which sees these initiatives struggle to keep up with innovation and therefore lag in adoption. In Table #REF below we offer a brief summary of file formats, which we identified as noteworthy during our research by their matching one or more of two criteria: 1) those that are frequently used in real-time 3D rendering applications and 2) those which are open standards. In Table #REF below, we give a brief description of the format, some preliminary notes on issues relating to their suitability for long-term preservation and, where applicable, point to their review in the Library of Congress' 'Sustainability of Digital Formats' resource.

Format	**Description**	**Preservation Notes**
Wavefront OBJ/MTL	OBJ is a simple, long-standing interchange format for 3D geometry. An OBJ file can be accompanied by an MTL file describing material properties.	While the most feature limited of the formats listed here (e.g. no support for animation), OBJ has proven long-term support across many applications. While the specification is public the licence is it released under is unclear and may be proprietary.

[21] https://www.loc.gov/preservation/digital/formats/

		LoC sustainability review: **https://www.loc.gov/preservation/digital/formats/fdd/fdd000507.shtml**
FBX	FBX is a proprietary format maintained by Autodesk and widely used as a 3D model interchange format in real-time 3D rendering applications.	Widely adopted in real-time 3D engine workflows. The binary format has been partially reverse engineered by Blender Foundation for their own import/export tools, but its proprietary nature remains a concern for long-term preservation.
X3D	X3D is a royalty-free open standard which can represent 3D objects and scenes. The format is an ISO standard and has been in development by the Web3D consortium since the early 2000s.	The format is not widely used in RT3D applications (neither Unity nor Unreal Engine 4 include X3D importers). Release of new versions of the specification moves slowly (the last ratified version, 3.3, was released in 2015) which means its features are out of step with those of other formats such as FBX and glTF[22]. **LoC sustainability review:** **https://www.loc.gov/preservation/digital/formats/fdd/fdd000490.shtml**
glTF	glTF is a royalty-free open standard designed for the efficient transmission of 3D models and scenes. It is maintained by the Khronos Group.	Primarily aimed at the web, adoption for real-time 3D applications seems likely if web-based RT3D continues to grow. Tools import/export support remains limited. Monitoring will be required given priorities regarding transmission efficiency and the potential for introduction of lossy compression features (e.g. Draco). **LoC sustainability review:** **http://www.loc.gov/preservation/digital/formats/fdd/fdd000498.shtml**
Alembic	The Alembic (.abc) format is a royalty-free open standard used to store complex animated 3D geometry. It is maintained by Sony Pictures Imageworks and Industrial Light & Magic.	This format is used in RT3D rendering and importing is natively supported in both Unity and Unreal Engine 4. As it only includes only the 'baked' animation, it is not lossless compared to source tools rendergraph.
Universal Scene Description (USD)	A royalty-free open-standard designed as an interchange format for complete 3D scenes composed of many elements. Maintained by Pixar.	Extent of adoption in RT3D is unclear, but both Unreal Engine 4 and Unity support import of USD scenes.

Table #REF. Summary of RT3D relevant 3D file formats identified during our research.

[22] https://realism.com/blog/gltf-x3d-comparison

Acquiring Real-Time 3D Production Environments

It is clear from our research that the complete toolchain employed in producing real-time VR artworks can be complex. This may be work carried out by a team and may involve the use of many distinct software packages; from the various authoring tools for asset creation to the real-time 3D engines used to bring them together. If the possibility of migrating or modifying VR software in order to achieve its long-term preservation (e.g. moving to new engine versions) is identified as a possibility for a particular artwork, it follows that as for other software programs (Engel & Wharton, 2014), production materials should be acquired and preserved. We are therefore left with the difficult task of trying to ascertain how best to approach this problem in the short term. We propose that the capture and archiving of production environments as disk images may offer a pragmatic approach. When captured from a computer system, we can consider the disk image a representation of a complete *software environment*, incorporating an operating system, installed programs and user data. This image is useful not only because it encapsulates all the important bits of data, but because we can use it as a basis for emulation or virtualisation as a means of accessing these environments in the long-term.

There are two possible approaches to this which have different strengths and weaknesses. The first is to disk image computers used in production directly. While providing the best way to ensure a complete capture, the indiscriminate nature of this approach results in the capture of all programs and data installed on the target machine, which may include unnecessary and/or personal data that would not be suitable for archiving. The second approach is to acquire the engine project (essentially a structured directory) and recreate the production environment on a physical or virtual machine by installing the appropriate engine binaries and testing the configuration. This is much less invasive than the first approach and would cut the amount of data captured. However, it also risks incomplete capture, particularly if appropriate engine binaries are no longer available with which to reconstruct. In either case, institutions would be advised to work closely with the artist in this process to ensure something useful is acquired. The decision over when it is necessary to maintain access to asset production materials is a more difficult one to resolve. it may be more suitable to maintain access to the original project files and software environment (including software used for modelling, texture painting, material definition etc.) in order to maximise the potential for future conservation treatments, although this may add significant overheads in storage and present challenges in terms of getting access to this level of information.

Best practices in software preservation currently indicate that disk images should be captured as raw physical images — complete encapsulations of the structure and content of a storage medium as a file (including boot partitions and other structures generally hidden from users). For cases where disk images cannot be created, reverse engineering techniques may provide one means of extracting a limited amount of equivalent information from application binaries — for example, tools exist to extract assets from packed data files. However, these may pose legal concerns through their violation of engine licencing terms and are unlikely to offer anything near a complete representation of the source project given the difficulty presented in decompiling native and shader code.

3.2. Real-Time 3D VR Display Materials

This section focuses on the software used in the display of RT3D VR artworks. This involves the software binaries engaging the software and hardware layers of an VR system, as described in Section 2 #REF, as frames are generated on-the-fly and sent to a display device. Works utilising RT3D are displayed by executing software — known as a *build* — in a suitable hardware and software environment.

Build Packages

A build is produced from a 3D engine project (see Section 3.1 #REF above) and usually contains a combination of executable code and packaged data arranged in a well-defined directory structure. In most cases this will be the primary media used in the display of the work, analogous to a master. Most builds produced by modern RT3D engines are compiled to low-level native code. Despite commonality among CPU architectures on contemporary computing platforms (Windows, MacOS and Linux all utilise the x86-64 architecture), this low-level code is highly platform specific due to the use of system calls, dynamic library references and other OS-specific features. A native build can however include support for an array of 3D APIs and VR runtimes, providing these are supported by the host platform. For most RT3D virtual reality builds on Windows, the dominant platform for native RT3D VR software, the build will consist of:

- At least one Windows Portable Executable (.exe) file
- Additional bundled DLL dependencies (e.g. Mono Runtime for Unity or the Ogg Vorbis audio decoder library)
- A set of packaged data assets (e.g. .pak files in Unreal Engine 4 and .resource files in Unity)

Analogous materials would be expected for other platforms. Modern game engines, including Unity and Unreal Engine 4, allow cross-compiling for the creation of builds for different target platforms such as Windows, MacOS, Linux and Android from the same source project.

WebGL RT3D uses a different set of technologies from native builds, being built as higher-level (non-native) code which is interpreted by a web browser at runtime, such as Javascript. The emerging WebAssembly language is an exception to this, which offers a low-level binary format which can be compiled to from languages like C++ and is executed in a virtual machine in the browser. In general, WebGL-based RT3D builds are dramatically divergent from native RT3D builds in that they much more closely resemble a website in structure. Assets are typically not packaged (rather, stored as OBJ or glTF files) while much of the code is present in a human-readable form. Other technologies provide further abstraction layers on top of WebGL to make content authoring easier, including AFrame and Three.js, although these are aimed at creators working outside of third-party game engines.

Acquiring Real-Time 3D Software Builds

A starting point for an effective build preservation strategy is to install and configure the software, ideally in collaboration with the artist, on a suitable set of hardware. Precedent in the care of software-based artworks at Tate is to create two of these reference systems for a specific artwork. In addition, it is important to acquire builds for as many different platforms as possible. Building for Linux in addition to the original target platform may be particularly advantageous. While VR support is currently limited in Linux, the open-source nature of the Linux ecosystem may make the possibility of recreating a suitable software environment in the future more feasible. Target VR runtimes also need to be considered at build time, as support for a specific API must be included in the platform-specific binaries. Again, ensuring builds with support for as many APIs and VR runtimes as possible is advised.

As for most compiled software, builds are not easily examined or analysed in any detail as code is compiled and assets are packaged in proprietary container formats. The means that there is not much that can be done to stabilise them at the file level once they are generated. Instead, preservation of binaries as-is will need to focus on environment-level approaches such as emulation and virtualization. We propose that in preparation for this work, and the potential failure of computer storage media, raw physical disk images are created for the storage media contained in any unique physical reference systems. The resulting disk images ensure the complete software environment is captured and can then be used as the basis of future emulation and virtualization work. At the time of writing, available tools provide limited support for the emulation and virtualization of contemporary VR systems. While we can boot contemporary operating systems such as Windows 10 in emulators, the limited capabilities of virtual graphics hardware makes achieving requisite performance levels impossible for all but the most rudimentary real-time 3D graphics. Furthermore, the inability to access GPU hardware using the native drivers means that direct mode access to the VR headset is inaccessible. Instead we are reliant on continued use of physical graphics hardware via PCIe passthrough.

3.3. Real-Time 3D VR Documentation

Given the large number of hardware and software components, the distinctness of user experience and external environment and the interactive nature of the medium, documentation is a potentially expansive topic in relation to real-time 3D VR. Our research has only really scratched the surface in terms of identifying how we might approach these novel documentation requirements. It seems likely that in the short term many existing tools and approaches from the fields of art conservation and digital preservation will be suitable to guide aspects of the documentation process. In particular, there is a well established precedent for documenting technically complex, installation-based artworks in time-based media conservation. We feel that in the areas of installation documentation and collection metadata, existing templates which have already been adapted for software-based artworks are likely to be suitable with minimal additional modification. In this section we will briefly consider the extensions required to existing methods that we have identified and propose areas for further research.

Acquisition Information Template

In the first phase of our research we identified that, as for any other artwork acquisition, we would need to gather sufficient information about that artwork to make effective decisions about bringing it into a collection, the archiving of relevant components and planning for its long-term preservation. Given the number of medium specific questions which arose in scoping this, we identified a need for a tool to guide the information gathering process at the early stages of acquisition and so developed a document template for this purpose[23]. This template is designed to be completed by or in close collaboration with an artist prior to receiving media from the artist. We tested this template by sharing a blank template with the artists we interviewed and inviting them to complete it for a specific artwork. Version 1.00 of the template is included in this document in Appendix 2. Further enhancement of the template has been carried out by Savannah Campbell and Mark Hellar and was presented at AIC 2019[24].

Understanding Performance and Rendering Characteristics

As discussed in earlier sections, each time a RT3D VR artwork is executed an entire system of hardware and software components is engaged. The rendering pipeline from data and program code through to output frames presents many points of potential variability, which should be identified and documented if we are to ensure consistent rendering in future iterations of the work with variable hardware/software configurations. In this section we will briefly introduce some of the key points of variability we have identified within the pipeline as a starting point for further research in this relatively unexplored area. This information will be particularly useful in identifying problems resulting from changes made to the software or its execution environment in the course of applying preservation techniques. We feel that a sufficient understanding of these characteristics to allow their effective documentation and management should be a long-term goal of the digital preservation community — one that will likely require considerably more research in the coming years.

Characteristic	**Description**	**Analysis & Documentation Approach**
Bit-depth	Bit-depth describes the possible range of values in which color/luminance can be expressed. Real-time 3D applications can support 8-bit or 10-bit per channel	Usually controlled with GPU driver utilities (e.g. NVIDIA Control Panel) but involves compatible OS components and suitable display equipment to be applied. Capture

[23] Currently available via the PIMG file share: https://groups.io/g/pimg/files/Acquisition%20Templates

[24] https://aics47thannualmeeting2019.sched.com/event/Iujq/electronic-media-from-immersion-to-acquisition-an-overview-of-virtual-reality-for-time-based-media-conservators

	output, and can use the full dynamic range of 0-255 (or 0-1023 for 10-bit) when connected to a suitable display device.	cards may provide an independent means of verification.
Colour Space	The colour gamut, gamma and white point for output frames. For real-time 3D applications this is usually sRGB, which can be converted by TV equipment to the similar Rec. 709.	Usually controlled with GPU driver utilities (e.g. NVIDIA Control Panel) but involves compatible OS components and suitable display equipment to be applied. Capture cards may provide an independent means of verification.
Framerate / Frametime	Measurements of the speed with which frames are generated. Framerate describes the number of frames created per second, while frametime is the amount of time taken to generate these frames.	It is not clear how existing RT3D monitoring tools such as MSI Afterburner / RivaTuner Statistics Server interact with VR runtimes. Runtime specific tools such as Oculus Profiler and SteamVR Frame Timing Tool should be used instead.
Latency	Measurement of the time taken for physical input to be translated into output frames.	We are not aware of any tools for measuring this for an arbitrary RT3D VR application, at time of writing.
3D API Feature Level	The set of shading related features supported by the 3D API, GPU and driver set, and targeted by a RT3D application.	We are not aware of any tools for identifying feature level support for an arbitrary RT3D VR application. Must be identified by examining the source project.
Internal Timing	The clock according to which events unfold within the simulation of the virtual environment.	Can only be identified from examining program code. This may cause issues where events are not coded to be framerate independent and thus become locked to processing speed[25].

Table #REF. List of performance and rendering characteristics and approaches to their analysis and documentation.

User-Perspective Video Capture

An effective way of capturing the experience of interacting with a real-time 3D VR artwork in a software-independent way is to capture the perspective of the user as video.

[25] https://www.construct.net/en/tutorials/delta-time-and-framerate-independence-2

User-perspective video capture then, is the recording of moving image frames as perceived by the user of the VR system through, for example, an HMD. At least two approaches can be taken to this kind of capture: fixed-view and 360 degree. Fixed-view video is captured from the perspective of the virtual camera as the user moves. Both rotational and positional movements become fixed to those that were carried out during the period of recording. This kind of video can be captured before or after the compositor carries out VR runtime specific processing. Pre-compositor capture is undistorted and without frame interpolation, while post-compositor may include the effects of these runtime processes.

An alternative approach is to capture 360 degree video from the perspective of the user, which captures the entirety of the users surroundings and so allows rotational tracking to be maintained in the resulting media. This is therefore interesting not only as a documentation technique, but as a preservation approach as it can offer a surrogate experience for VR experiences which do not use positional tracking (also known as 'on-rails'). Furthermore, the 360 video version does not have the fixed dependency on a specific real-time 3D rendering technology, as the output of the capture process is simply video data which can be played back in a variety of players (see 360 Video #REF). Features for capturing 360 video from real-time 3D environments are present in both Unreal Engine 4 and Unity but we are not aware of any third-party tools for capturing 360 video in compiled real-time 3D applications at time of writing. The primary limitation of the 360 video capture approach is of course that positional tracking and other forms interactivity beyond rotational tracking are lost. Additionally, a byproduct of bypassing the normal fixed field-of-view player camera is that certain 'screen space' visual effects will not be captured (e.g. vignetting, light shafts, motion blur)[26].

Capturing both fixed-view and 360 video in a real-time 3D engine is resource intensive and can generate very large volumes of data if captured in an uncompressed form. Strain on the host system can be eased by utilising dedicated encoding hardware (e.g. NVIDIA's NVENC feature available on some GPUs), freeing up the GPU to operate unimpeded. Decisions over appropriate video encoding are similar to those when working in other video contexts (e.g. chroma subsampling, bitrate, compression), but care should be given to ensure encoding matches the frames output by the GPU as closely as possible for an accurate capture (e.g. colour space, bit-depth, framerate). This is impeded by the lack of transparency over where software tools actually capture frames in the rendering pipeline, a problem which may be solved by using dedicated video capture cards. Further research is required to better understand how we might effectively carry out this kind work.

The missing component in capturing user experience is the nature of interaction as it occurs within the physical installation space. Understanding how interaction has occurred in the past offers an important contextual insight into how that work was presented and received. While photographic methods of documentation will be successful in capturing VR artwork installations to some extent, the dynamic and interactive nature of VR points towards the importance of video as an extension of this. Captured simultaneously with user-perspective

26 https://www.unrealengine.com/en-US/tech-blog/capturing-stereoscopic-360-screenshots-videos-movies-unreal-engine-4

documentation this provides one way of connecting the users virtual experience with their movements in physical space. Tests during a hackathon at iPRES 2019[27] demonstrated that capture tools such as Brekel OpenVR Recorder provide one way documenting this kind of interaction, although further research is required to understand how the outputs could be used.

4. 360 Video

4.1. 360 Video Production Materials

360 video can be produced from two distinct workflows, as a capture from a real-time 3D engine (see 360 video capture #REF) or captured from a camera or array of cameras. Camera captures can be monoscopic or stereoscopic. Monoscopic 360 video is captured from a single point of view, with the single image repeated in both eyes, and so provides no illusion of depth. It is typically captured with dual fisheye lenses, arranged back to back, each providing 180 degrees of view. Stereoscopic 360 video employs an array of lenses to generate images which provide a slightly different point of view for each eye, which has the result of creating the illusion of depth when viewed through a HMD, through the principle of parallax.

In both of these examples, raw video data from the camera captures have a geometry that must be *stitched* together to produce viewable video content. There are several software programs that undertake the process of stitching, and often a commercially available camera is bundled with its manufacturers proprietary software for achieving this. There are also third party options available, which contain templates of commonly used lens arrays and in some cases can automatically detect camera laouts. Given that overlapping pixels are blended and discarded, the process of stitching is an inherently lossy one. It looks likely that stitching algorithms will improve in accuracy with time and increasing processing power, pointing to a strong case for archiving raw camera footage for improved stitching in the future for preservation purposes. This has possible implications in terms of vast increases of storage required for this raw footage, particularly in the case of multi camera stereoscopic capture.

4.2. 360 Video File Types

Data resulting from the stitching process is stored in a planar video format. The method employed to represent the 3D information in a planar video file is referred to as the *projection format*. The most common is called *equirectangular* projection, familiar to us as the method used to display the surface of the earth in a 2D atlas. This projection type inherently has curvilinear distortion in the stored video file, causing inconsistent scaling of features across the image. In the case of a 360 video file this means that the top and bottom of the image use a disproportionately large area of pixels — a varying *pixel density* — an

[27] https://ipres2019.org/static/pdf/iPres2019_paper_154.pdf

inefficiency compounded by the combination of this area of the image with highest pixel density being in the viewers peripheral vision. This problem has led to the use of *cubemap* projection, where the 360 information is stored in a form which can be radially projected onto the inside surface of a cube. This projection type has more even pixel density than equirectangular projection, though density still varies over the cube faces. Further benefits come from efficiencies in compression for playback files — temporal compression assumes that objects within the frame move in straight lines and therefore aren't as efficient in compressing media with curvilinear distortion, where a straight line in the physical world would be appear curved in an equirectangular file. A further development in projection is the equi-angular cubemap (EAC) — in this projection type, a distortion mesh (#REF) is introduced onto each face of the cube, resulting in each degree of viewing angle being assigned an equal number of pixels.

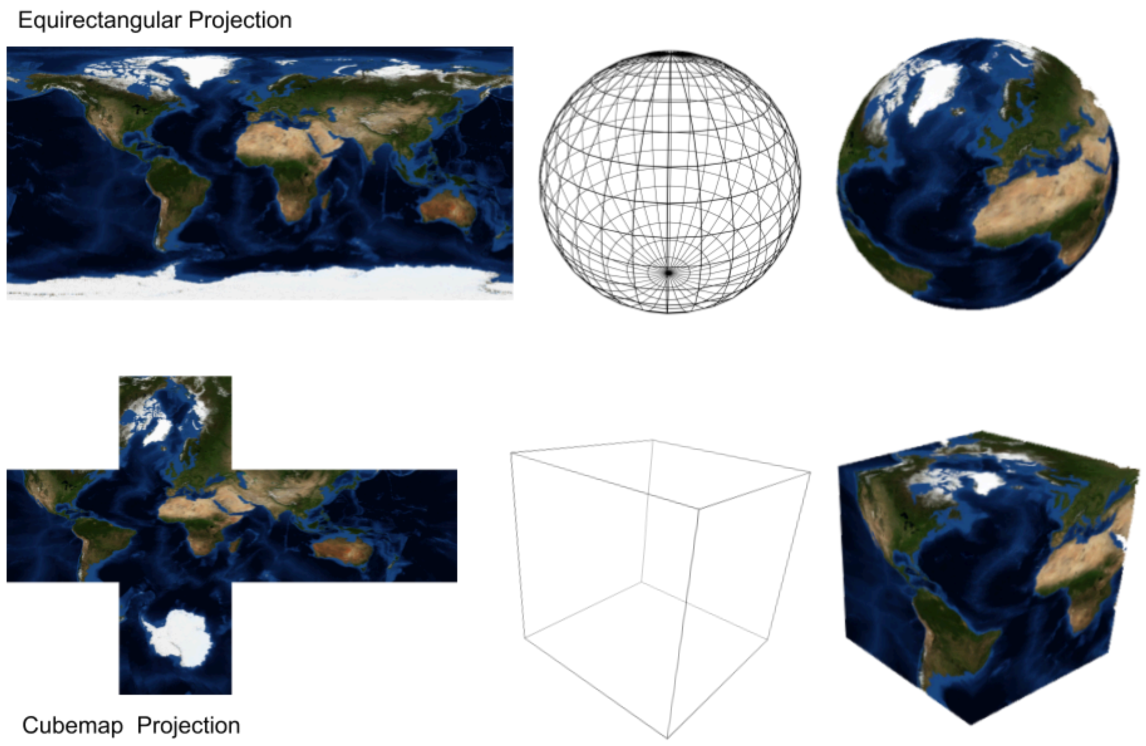


Figure #REF. Representation of equirectangular and cubemap projection types, moving from unwrapped image (left), associated 3D geometry primitive (centre) and finally the images projection onto this surface (right).

In online platforms where streaming efficiency is prioritised, there are other projection types being implemented with a focus on performance. Facebook have developed a pyramid projection system which gives priority to information in the users direct vision at any given time, whilst minimising data use in the periphery and behind the user's vision. An advantage that the cube and pyramid maps have over equirectangular in this scenario is that the different faces of the cube can be layered with different amounts of compression — the video on the periphery can be streamed at a lower quality without apparent loss from the users perspective, bringing further bandwidth efficiencies.

360 video file formats show similar characteristics to those of planar video. The aspect ratio of monoscopic video is commonly 2:1, which relates to 180 degrees of vertical vision and 360 degrees in horizontal. The resolution of the stored file can be higher than that seen in planar video — considering the viewable area of the video is 90 degrees and hence around 25% of the entire image, a 360 video would have to have four times the resolution of a planar video to achieve comparable levels of pixel density. The high resolutions of data required are pushed further where video is stereoscopic, as images for each eye are stored either in a side by side (SBS) or over/under (OU) format. This requires a further doubling of the image resolution, meaning that to view a stereoscopic image with comparable quality to 1920x1080 in two dimensions, a horizontal resolution of 16000 pixels would be required in an SBS file.

360 video files are played back using specialised pieces of software known as 360 video players. These players are essentially real-time 3D software (see Section #REF) capable of decoding video and include a renderer capable of projecting video frames into 3D space. In playback, the 360 video file is projected onto the inside of a primitive 3D object which corresponds to the projection format chosen (e.g. a sphere for equirectangular or a cube for cubemap). A UV map (see Section #REF) or equivalent information specific to this projection format tells the 360 video player how to correctly distribute the video frames over the surface of the 3D object (#REF). Players may be standalone pieces of 3D software or may be authored using RT3D engines such as Unity or Unreal Engine.

As for other forms of VR, a further requirement for 360 video files is a minimum frame rate — of 30 (FPS). Lower frame rates than this cause visible lag when moving your head from side to side in an HMD, with lag considered to be one of the major causes of motion sickness within VR. High frame rates, large resolutions and stereoscopic images all combine to cause a significant increase in file sizes compared to planar video in similar formats. As a result, compression is often employed in playback to address issues of streaming bandwidth and storage — especially in the case of untethered devices such as the Oculus Go which is limited to 32GB or 64GB of onboard storage. At the time of writing, there are no specific codecs for 360 video — H.264 and H.265 are commonly employed.

4.3. 360 Video: Audio

Audio accompanying 360 video can be either stereo, where audio is played through a user's headphones irrespective of the position of the head, or it can use the position of the HMD to calculate a binaural mix from a multitrack spatial recording, relative to the position of the head.

In the case of 360 video captured from a camera with accompanying spatial audio, it is likely to be recorded by an ambisonic microphone — which in its simplest form (first order, or 4-channel) uses an array of four capsules to record 360 degrees of audio. This audio in its recorded form is referred to as ambisonic *A format*. For this audio to be integrated into a video file, it undergoes processing to generate *B format*, where the raw audio from the capsules is converted to four audio files that represent the X, Y and Z spatial axes and W, overall amplitude respectively. B format represents the captured space or *sound field* in an abstract form- an important principle is that it is *speaker agnostic* and can be decoded for a variety of speaker arrays. In VR, it is decoded by a *head-related transfer function* (HRTF) to generate a binaural mix for listening. An HTRF is an algorithmic process that models a human head and ears within a sound field recording, and approximates what a pair of human ears would hear within that space. It employs several techniques to do this, such as calculating the time difference taken for a sound to reach each ear, calculating the diffraction of sound waves caused by the head, and the sound waves heard through being absorbed into the head. This process is undertaken in real time by the player, taking positional input from the HMD and generating a binaural mix down in relation to this, allowing a more immersive listening experience than stereo. Several data sets for the HTRF algorithm exist, and some allow for adjustment of virtual head size and ear spacing, though ultimately they are data sets based on averaged human dimensions and ear shapes, therefore have finite accuracy. The conversion from A to B format is unique for each microphone, based on microphone make, model and individual capsule calibration. Software to undertake this process (often a VST plugin for a digital audio workstation) is therefore provided by the manufacturer. Some microphone arrays are available that record natively into B format, though these are less common. Second order (or 9-channel) ambisonics employs the same techniques but achieves improved spatial resolution with 9 microphones, while third order (or 16-channel) ambisonics uses 16 microphones and so on up to much higher channel numbers.

Care must be taken in the preservation of ambisonic audio to note the specific conventions used to generate the B format files, as these impact how it is played back. B format files can be placed in different orders according to various conventions such as Furse-Malham, ACN and SID. Furthermore, to achieve the correct spatialisation the files are normalised in relation to each other according to various conventions such as maxN or SN3D. Two prominent exchange formats exist, FuMA (**Fu**rse-**Ma**lham) prescribes the channel ordering WXYZ and maxN normalisation, whilst AmbiX (**Ambi**sonic e**X**change) prescribes WYZX (ACN) ordering SN3D normalization. Video containers are agnostic to these conventions and we rely on file metadata to accurately reflect the scheme used to ensure that a playback device is able to correctly interpret the files.

4.4. 360 Video Metadata

Due to the increased number of variable parameters in 360 video over planar video, there are increasing demands on the file metadata to accurately describe projection type, distortion map, and audio convention. There is a spatial media metadata standard put forward by Google, and a corresponding spatial media metadata injection tool[28]. One particularly promising aspect for preservation is that it proposes to describe the relationship between video file and projection type mathematically. At the time of writing it is too early to comment on how widely it has been adopted. In absence of a metadata standard, many

[28] https://github.com/google/spatial-media

players interpret a string of characters from the file name- such as “_LR.mp4” for Left/Right stereoscopic panoramic video.[29]

4.5. 6DOF & Volumetric Video

One of the primary differences between 360 video VR and RT3D VR is that in 360 video playback interactivity is limited to rotation and tilt of the head. Moving the head from side to side or up and down does not result in any change in view, given that the camera’s position in a space is fixed as the video is pre-recorded from a fixed perspective. The ability to move in any direction is referred to as *six degrees of freedom* (or 6DOF).

There are several attempts underway in the VR industry to make this possible for 360 video, such as Adobe’s project Sidewinder or HTC’s 6DOF Lite. Some of these methods make use of a depth map which can be generated in some cases by stitching programs or capture hardware, whilst others, such as HTC’s 6DOF Lite are able to generate a limited 6DOF experience from existing stereoscopic video by generating depth information in real-time. Tools to perform these functions are blurring the boundaries between 360 video players and real-time 3D engines, as more complex features of RT3D rendering such as vertex displacement are employed.

This blurring with RT3D rendering is taken further still by volumetric capture, a technique that uses an array of cameras to capture a scene from every angle, hence allowing a subject or *volume* to be viewed from any angle. It differs from common 360 video formats in that it is often filmed from the outside and might be used to capture the performance of a human for example. It has similarities to the techniques used in photogrammetry, a capture technique which can derive a 3D representation of an object using photographs taken from multiple perspectives, but is carrying out this process of conversion from still images frames to point cloud 3D data in real time. At the time of writing, the amount of storage and computational power required put it outside the scope of this paper.

5. Suitability of Existing Preservation Strategies

In this section we will explore how existing preservation strategies might be applied in the long-term preservation of immersive media artworks, with the aim of identifying their suitability and limitations. In Section 2 we highlighted the complex set of interrelated components which make up a generic VR system. As we would expect for any other set of technologies involved in a complex time-based media artwork, the primary risk to the preservation of these systems is the obsolescence of the hardware technologies employed and the impact this has when hardware components inevitably fail. This has a trickle down effect on the software components, which can be preserved at the bit-level using

[29] https://developer.oculus.com/documentation/native/android/mobile-media-overview/

well-established approaches to archival storage[30] but become impossible to execute when components of the hardware system they connect to fail. We can therefore characterise VR systems as fragile: changes to any one component can result in complete failure of the system. To give a concrete example: the failure of HMD hardware (which is likely to be accelerated as a result of public gallery use) which cannot be replaced or repaired may force the use of new hardware, which results in the loss of the distinctive character of the original HMD (e.g. the low resolution panels and limited field-of-view present in early models) at best, or at worse the complete loss of functionality due to the replacement HMD lacking software support for communication with the VR application (e.g. the likely scenario that a newer HMD requires use of new drivers, APIs or runtimes).

For RT3D applications and 360 video players, these risks can to some extent be lowered by making decisions at the time the software is built which allow for maximum compatibility across different platforms. For example, building a version of the software application with support for a variety of VR runtimes or building multiple versions with support for operating systems and 3D APIs. Carrying out this kind of preparatory work allows us much more flexibility in recreating a suitable execution environment in the future by maximising the ways in which a functional system could be pieced together. If an open standard such as OpenXR is widely adopted and backwards compatibility is retained, building in support for such a standard over proprietary solutions will also be a significant benefit. However, at the time of writing we cannot be certain as to how the technology will develop and must instead consider our options for more interventive forms of preservation.

Using the well-established concepts of storage, emulation and migration as a starting points (#REF here?), in the following sections we will consider for each of these how it might be applied to preserving virtual reality artworks. We do not propose that these approaches are branching paths, but rather that they are ways of grouping related techniques which could be used in tandem to maximise chances of long-term preservation.

Storage Approach

The first and most obvious response to the primary risk of hardware failure is to simply securely store digital objects and stockpile suitable hardware. While we acknowledge that this is theoretically an effective strategy, particularly in the shorter term, in practice it poses many challenges. Stockpiling has a precedent in time-based media conservation, and alongside ongoing collaboration with specialised communities outside the museum, is one of the primary strategies for ensuring long-term access to CRT monitors for Tate’s large collection of video art. However, applying the same logic to VR system hardware raises many difficult to answer questions. Given the short lifespans of interactive equipment used in public galleries and Tate’s mandate to care for artworks in perpetuity, how many pieces of hardware is enough? Given the relatively small number of VR artworks likely to be acquired by any one institution (this remains an emerging medium), how can we justify the

[30] As issues of effective archival storage are generic across digital materials, these will not be addressed here. It is worth noting however that virtual reality artwork binaries and project files, as well as disk images, can present very large data volumes.

considerable financial outlay of acquiring such a quantity of hardware? Given that VR-specific hardware like HMDs and tracking systems are contingent on specific computer hardware, would be required to stockpile these systems? This activity then quickly begins to outgrow what would be (both in financial and maintenance terms) practical for any one institution to manage. Clearly stockpiling for the very long-term is not a suitable preservation strategy, and for now we recommend following Tate's existing best practices of acquiring two sets of reference hardware.

Emulation and Related Approaches

Emulation and related approaches are those preservation strategies which focus on maintaining a suitable execution environment, made up of hardware and software components, while leaving the primary digital objects (i.e. the software build or 360 video player and video data) unmodified. This includes not only well known digital preservation techniques such as emulation and virtualisation, but also the use of related tools like compatibility layers and wrappers.

Emulation involves the use of tools which recreate a particular set of hardware (usually centering on a particular processor type) in software. In preservation terms, this allows the software environment to be separated from physical hardware (usually as a disk image) and executed on emulated hardware. Related to emulation is virtualization, which involves many of the same principles but additionally allows the guest (i.e. the emulated machine) access to some physical hardware on the host machine. Emulation and virtualization have demonstrated uses in digital preservation [add #REFs here] and has recently been integrated as a standard part of acquisition workflows for software-based artworks at Tate. It therefore would appear to be low hanging fruit in terms of its applicability to preserving VR artworks, particularly in comparison to the more involved work required to migrate software (see following section).

At time of writing, full system emulation does not appear to be a feasible strategy for preserving virtual reality artworks. Taking this approach, which completely removes dependency on physical hardware, software emulates graphics hardware in software and runs the code on the emulated CPU. This has the effect of massively reducing its capabilities in comparison to dedicated GPU hardware. At time of writing, the emulated GPUs we examined (including those packaged with QEMU, VirtualBox and VMware) do not offer sufficient 3D rendering capabilities to run virtual reality artworks at required speeds. Furthermore, the current generation of VR runtimes require access to a physical graphics card in order to access the HMDs in direct mode. The workaround for this is to allow the emulated environment a level of access to a physical graphics card via paravirtualization or passthrough, but this does not remove dependency on this physical hardware and the associated software stack, thus limiting the value of the resulting emulated machine for preservation purposes.

The use of compatibility layers and similar tools offers an alternative to full system emulation with short term applications in preservation. Compatibility layers (sometimes called wrappers) intercept system calls made to one API by a program to another. By virtue of shared processor architecture (most desktop computing software is compiled for x86-64 processors) a program built for one operating system can be executed on another operating with minimal performance overhead, as any native code can be run on the CPU as-is. For example, Wine[31] allows the execution of software designed for modern Windows operating systems on modern Linux operating systems, including applications which require the use of the Direct3D API, by translating system calls from one platform to another. While the dependency on a specific processor architecture remains, tools such as Wine do have the potential to at least slow obsolescence. Compatibility layers are also emerging to allow the use of applications designed for one VR hardware set with another. For example, ReVive allows the use of a HTC Vive hardware with applications designed to only support the Oculus Rift hardware. Whether these will emerge to support legacy VR applications in the future is of course impossible to predict, but given interest in VR from the gaming community it's easy to imagine retrogaming enthusiasts taking on this kind of challenge. The use of proprietary technologies will of course hamper progress in this area, giving further support to the notion that we should favour open standards where possible.

While using these tools to display works seems unlikely in the short term, due both to their formative state and a lack of need as suitable physical hardware remains available, there are important steps we can take now to prepare for their use in the future. To prepare for the use of emulation, we can create raw disk images of the storage media of artist-verified and tested computer systems, ensuring that the complete software environment is preserved at the bit-level. To supplement this and ensure we can match appropriate emulated software in the future, the hardware of the physical computer system should be carefully documented, particularly the connections between specific hardware and software components. Finally we should continue to advocate for the use of open standards in the development of VR systems where possible, including support artists in creating new software builds where possible. Where proprietary technologies are unavoidable, we can try and advocate for greater openness within the industry through new connections and ensure that legal provisions better predict those reverse engineering these technologies in order to support legacy access, such as the recent DMCA copyright exemptions passed on the US (#REF).

Migration and Related Approaches

In contrast to emulation, migration and related approaches are those which modify the *digital object* (i.e. the software build or 360 video file/player) in order to keep it running in contemporary technical environments, rather than the environment. This corresponds to well known digital preservation strategies such as migration and less well explored approaches like incremental maintenance. Adapting the digital object in this way is useful for long-term preservation because it would allow us to create new software builds with support for future hardware and software. For example, we might add support for a new VR runtime API or recompile native code so that it can be run on a different processor architecture.

[31] https://www.winehq.org/

Migration in software preservation would typically involve rewriting the underlying code and recompiling to create new software builds. Modern game engines are very large and complicated pieces of software, developed by large teams over many years and usually only made available under restricted licences which may prevent modification or reuse of the engine outside of specific circumstances. In the context of real-time 3D applications, a complete code rewrite is therefore highly unlikely to be viable in the foreseeable future due to the resources that would be required to carry out such a feat and the associated legal issues. Instead, we might consider migration as theoretically viable *between* game engines, a process which would involve asset or scene migration and manual reconstruction of other dynamic elements such as scripting if the programming languages used differed.

The first hurdle to achieving is that this would require access to complete production materials, including all the data assets used in an exportable form and the entire engine project and associated binaries. As discussed in Section #REF, there are currently barriers to this due to the need to acquire complex production environments which are challenging to stabilise as something which can be preserved within existing digital preservation frameworks. Even when applying disk imaging, we remain dependent on the unverified suitability of in-engine export tools to ensure we can retrieve any data contained therein losslessly. Even where this is possible, it becomes very difficult to ascertain whether the characteristics of original software, which are the product of the unique renderer implementation of that particular engine, could accurately recreated in another engine. This is not a question we can answer at this time, as there are no case studies that we are aware of nor can we predict the direction the development of real-time 3D software will take. We suspect that relevant knowledge may exist within the video game industry, where porting (essentially analogous to migration) has been a common activity although there are only very few examples of the details of such processes being discussed in public forums[32]. Future research is definitely required, which could draw on the previous generation of VR technologies from the 1990s as case studies for migration to current generation hardware.

An additional question is when to carry out this kind of work. In a typical time-based media approach, this would involve general monitoring of technologies for obsolescence and migrating to new technologies as appropriate, usually heavily influenced by instances where an artwork is realised as an installation. The frequency that this would be required is not easy to ascertain based on our existing knowledge of this emerging technology. However, if HMD models are being replaced by new models at an interval of roughly three years (see Section #REF), it is reasonable to suggest that obsolescence may occur over such an interval. This would rapidly accelerate the regularity of intervention required to keep software in step with the current technological environment. This invites consideration of an alternative approach to object adaptation that may be highly effective in the short term: incremental migration. This would again require access to complete production materials, but instead of focusing on radical intervention after long periods of time as-in migration as described above, it would focus on the short term by incrementally updating the source project as new engine versions are released. We are not aware of any case studies of this

[32] E.g. https://www.gamasutra.com/view/news/222363/What_exactly_goes_into_porting_a_video_game_BlitWorks_explains.php

kind of work being carried out and given the divergence from the prevalent reactive notion of digital preservation (particularly the increased resources required), it is very hard to draw any conclusions of its viability at this time. Again, further research is required to determine whether a framing of migration as maintenance would be practical or desirable in the cultural heritage sector.

Irrespective of the approach we take, there are certain essential steps that can be taken to prepare for object-centric approaches. The most obvious of these is to ensure that production materials are acquired where possible, and that all the necessary dependencies to open and build from these the source projects are also acquired. As discussed in Section #REF, disk imaging appears to offer a suitable solution to capturing production environments, with some caveats. In addition to this, the cultural heritage sector needs to consider the possibility for scenarios where there is a long delay between completing production and artworks being acquired by institutions. This points to a need for third-party tools such as engines, operating systems, VR runtimes and other dependencies to be independently preserved by suitable bodies. While this work may be occuring within the industry, this is not clear from available evidence. What a body outside of the industry might look like, or whether it might already exist is also unclear and requires further collaboration within the cultural heritage sector to determine, as well as new connections to be forged with the industry.

6. Summary of Recommendations

In this section we offer a set of recommendations for artists and institutions who are dealing with the immediate problem of caring from VR artworks. These represent a snapshot of our understanding of this topic at this time and we hope will be refined and built upon by others. With that in mind, we also provide a set of recommendations for future research topics in this area.

Recommendations for Artists

For artists we recommend the following steps are taken as a short term stabilisation strategy for the VR works they are caring for:

- Ensure you have a complete offline VR system (including hardware and software) configured and correctly running on the target hardware.
- Capture and archive a disk image(s) of the contents of the primary storage volumes of this computer system.
- Ensure you have a backup of this system (hardware and software).

For VR artworks with a real-time 3D components we recommend the following additional steps are taken:

- Create software builds for as many suitable platforms as possible, test them and archive these with configuration instructions.

- Maximise application support for a variety of VR hardware by using all suitable VR plugins and SDKs in the software builds created.
- Carefully manage engine projects and assets so that they are contained within a single location.
- Archive snapshot(s) of a production environment (typically consisting of at least configured game engine binaries and project files), ideally as a disk image.
- Use version control software to manage, track and document any further development or modification.

For VR artworks with a 360 video component we recommend the following additional steps are taken:

- Consider archiving raw camera output to allow footage to be re-stitched in higher resolution as technology progresses.
- Consider archiving the complete production environment for re-export, ideally as a disk image.
- In the case of works exported from RT3D engines, consider archiving the production environment, including project files and engine binaries, ideally as a disk image.

Recommendations for Collecting Institutions

For collecting institutions we recommend the following steps are taken as a short term stabilisation strategy for the VR works they are caring for:

- Ensure you have a complete offline VR system (including hardware and software) configured and correctly running on the target hardware.
- Capture and archive a disk image(s) of the contents of the primary storage volumes of this computer system.
- Ensure you have a backup of this system (hardware and software).

For VR artworks with a real-time 3D components we recommend the following additional steps are taken:

- Acquire or create software builds for as many suitable platforms as possible, test them and archive these with configuration instructions.
- Maximise application support for a variety of VR hardware by using all suitable VR plugins and SDKs in the software builds created.
- Acquire or recreate a production environment (typically consisting of at least configured game engine binaries and project files), ideally as a disk image.
- Use version control software to manage, track and document any further development or modification.

For VR artworks with a 360 video component we recommend the following additional steps are taken:

- Attempt to play the video file on a variety of different software players and untethered headsets as reasonably possible, identifying any variances in audio or video.
- Verify that the metadata correctly describes the projection format, distortion map, and audio convention.

- In the case of works captured from camera, consider archiving raw camera files to enable re-stitching at higher resolutions.
- In the case of works exported from RT3D engines, consider archiving the production environment, including project files and engine binaries, ideally as a disk image.

Recommendations for Further Work

For VR artworks in general we have identified the following priorities for further work in this area:

- Monitor development and support adoption of open standards for VR.

For real-time 3D VR artworks we have identified the following priorities for further work in this area:

- Monitor development and support adoption of open standards for real-time 3D software.
- Support further research into the practicality of maintenance as a preservation strategy, including how frequently maintenance would be required.
- Support further research in understanding variability in real-time 3D rendering, and the effective documentation and management of performance and rendering characteristics.
- Monitor and support the development of emulation and virtualization and their support real-time 3D rendering.
- Support further research into 3D formats for stabilising 3D model assets.
- Support further research into better understanding how to effectively capture fixed-view and 360 video from real-time 3D VR artworks.

For 360 video we have identified the following priorities for further work in this area:

- Monitor the evolution of metadata standards and their adoption.
- Monitor the evolution of projection formats and the implications for player compatibility and sustainability.

Bibliography

Campbell, S., 2017. *A Rift in our Practices, Toward Preserving Virtual Reality* (Doctoral dissertation, Master Thesis at Moving Image Archiving and Preservation program at New York University. https://www. nyu. edu/tisch/preservation/program/student_work/201 7spring/17s_thesis_Campbell. pdf).

Cranmer, C., 2017. *Preserving the emerging: virtual reality and 360-degree video, an internship research report*. Netherlands Institute for Sound and Vision.

Ensom, T., 2019. *Technical narratives: analysis, description and representation in the conservation of software-based art* (Doctoral dissertation, King's College London).

McConchie, J. 2018. *VR tools as spatial documentation.* Presentation at AIC Annual Meeting 2018.

Appendix: Tate VR Acquisition Template

[To be appended here?]