Back in the early days of personal computing there were very few standards for how most programs would run or would be coded for compatibility with later systems. IBM had a program operating standard, from which clone systems like the Tandy 1000 copied. Others, such as the Apple IIs and the later Macs used a completely different operating standard. In the business sector, specific programs were being coded for use on large scale mainframe systems. Technology was increasing rapidly in those years and there were huge jumps in performance of both PCs and Apple products even within a single year. If anyone wanted to run older programs or games that were coded for a processor running at a slower clock speed, they would normally be out of luck. Fortunately, many OEM companies caught onto this trend and offered turbo buttons on their computers to lower the clock speed of their 486 and Pentium processors, to accommodate the hardcoded demands of older programs and games created for DOS. This worked until the new millennium, when the Turbo button disappeared due to the birth of the GUI-based Windows PC, which no longer required users to access DOS. The newer processors which were released with Windows could no longer be underclocked enough to run the older programs without running into trouble with the operating system’s requirements. Later, graphics cards were standardized to AMD and NVIDIA after 3dfx Interactive filed for bankruptcy. The result was that those graphics driver calls (known as Glide) that a lot of programs depended on back in the late 90s (and even some after the new millennium) were made obsolete. Now older enterprise-grade software and consumer games no longer work on the freshly installed Intel Core i3, i5 or i7 that came with the newfangled Windows 2000, later operating systems and their prerequisite hardware. Enter the compatibility layer, a piece of software or an application program interface (commonly called an API) or wrapper which tries to make these older programs run as closely as possible to their original intent on modern hardware and operating systems.

According to the Emulation General Wiki, compatibility layers are programs which translate the system and API calls made by the native application to the appropriate calls on the host system regardless of the hardware or software configuration of the system it is now running on. This not only allows older programs from the same operating system family to run on the modern equivalent operating system it was designed for, but it can also be used or configured to allow cross-platform compatibility between different operating systems, as well as hardware. Compatibility layers differ from emulation as emulation, is the underlying hardware and operating system being emulated, or mimicked, to support the native software being run on any other system. [1]

Think of compatibility layers as a software middleman that translates these deprecated or cross-platform calls in a very lightweight program compared to a complete software emulation approach. Thus, at least in theory, programs running through compatibility layers should have no performance runtime hit nor loss of accuracy from the system it was designed for. Unfortunately, this notion is seldom the case in practice. Most compatibility layers have a noticeable performance overhead that has a negative impact on the operation of the program as opposed to the performance of the same program if the right configuration of hardware and software was present for that program to run natively. These performance hits are mostly caused by graphics API translations for non-native platforms of similar technology (such as the lack of Direct 3D / Direct X on Linux) and CPU instructions from a family that is the opposite to what the program was designed for (such as trying to run an Intel based game on AMD). There is also the occasional hiccup in accuracy in these layers as some of the calls may be untranslatable and thus have problems running the program fully natively. In games, for example, this usually results in trouble rendering the characters on screen creating visual artifacts, a completely unplayable experience, or a game that won’t start up in the first place. To counter this problem, some compatibility layers have components of software emulation built in to support different system architectures that are so unique that it is impossible to translate the calls directly, but instead emulate the architecture of the compatible system, relative to the program.[1]

In the case of compatibility layers for graphics cards, such as 3dfx Voodoo cards with the Glide API or the Direct X API being translated into Vulkan or Open GL calls for Linux, a wrapper is used. It is basically a drop-in replacement for deprecated driver libraries for older software and incompatible libraries for other operating system platforms, though some wrappers may require configuration prior to use. These wrappers mainly exist because neither NVIDIA nor AMD included the 3dfx Glide API driver in their native drivers for their graphics cards. The only other way to play the games or use software that depended on the Glide API was to wait around for a port to a modern API by the developers (which seldom happens); these wrappers are used instead. [1]

For consumers and my own interests, compatibility layers represent a radical idea to allow compatibility for older games and software that used obsolete graphics driver calls, allowing these games to be preserved so they may run as they did originally. For businesses, compatibility layers represent a godsend for trying to keep outdated programs that are still useful within the company running smoothly on more modern hardware. The IT field would likely see these layers as a way to facilitate running older programs that may be more reliable than their modern program equivalents. Although software and hardware emulation represents a huge margin in the retro gaming community and to an extent in the business sector, compatibility layers are the first and most cost-effective step to making these older programs work on newer systems and thus are one of the primary stops of many consumers, software developers (who are maintaining an old system) and companies who employ the use of ancient programs requiring outdated system software.

First, let us talk about a few examples of compatibility layers and how each of those pieces of software are unique. When in an ecosystem like the Microsoft Windows Operating System, many consumers expect older software to work out of the box; otherwise known as backwards compatibility. However, it is common for these new operating systems to not have 100% compatibility with older software. This is due to the exponentially increasing complexity of the operating system including the bug fixes and security holes patched in the new system upon which older programs might have come to rely. [2] The initial compatibility layer that all Windows users try is the built-in compatibility layer that exists for Windows. This compatibility mode is based on a shim-based approach where the shim acts as middleware between the program meant for an older version of Windows and the current Windows version. In fact, the implementation uses dynamic linked libraries or DLLs to have the outdated machine apps run natively using original code from previous operating systems. Based on the GUI interface to enable compatibility mode, Windows takes the version to imitate and executes code that makes the current operating system behave like the Windows version specified for the application. [3] This approach maps one interface to another or, in other words, one operating system (the current OS) to the other (the selected older OS). The static mappings of a shim-based system are a problem because they do not react very well to program targeted objects that do not have a defined type but still provide the functionality the program needs. [4] Because of these shortcomings, compatibility mode wasn’t very reliable in the early days of the Windows NT-based operating systems; hence Microsoft eventually released an Application Compatibility Toolkit as a workaround while they worked on a better solution. The documentation on this Toolkit continues to be updated, with the last edit being made in 2018. [5]

WOW or Windows on Windows is the current implementation of this shim-based system, released for Windows 10. This is a subset of the original compatibility mode which, starting in 1993, allowed 16-bit programs in DOS to be converted to 32-bit applications and run natively. This subsystem was included for all 32-bit Windows NT-based versions (from Windows 2000 onward) where originally it was called NTVDM (or New Technology Virtual DOS Machine). With Windows 10 and other 64-bit versions of Windows (Windows 7 being the most prominent), this functionality had changed to 32-bit applications being converted to a 64-bit compatible version for Windows and thus named WOW64 or Windows on Windows 64. The compatibility for WOW64 is vast and is vetted to have most and possibly all programs from 32-bit Windows work natively in 64-bit mode. Unfortunately, the commercial and consumer switch to the x64 based Windows operating system had an unexpected casualty. Microsoft scrapped the NTVDM implementation which axed compatibility for DOS programs in 32-bit mode. Emulation General Wiki suggests using the Emulator called DOS Box or a subset of the compatibility layer WINE called WineVDM based on NTVDM which Microsoft discarded. That being said, compatibility for 16-bit applications isn’t very smooth on some of these implementations of WOW in 32-bit Windows including the original NTVDM. In fact, some games ran too fast or came with visual glitches when run in 32-bit mode via WOW. [7]

WINE (Wine Is Not an Emulator), is one of the most popular compatibility layers. It enables Unix-based systems to run Windows programs natively. It is compatible with MacOS, BSD and various distributions of the Linux operating system. It works by translating Windows API calls into POSIX calls which is the Unix equivalent API. It was originally developed in 1993 by Bob Amstadt to run Windows 3.1 Applications on Linux. Development of the compatibility layer is currently under the direction of Alexandre Julliard. After 15 years of active development, the first stable build was released to the public in 2008. The majority of the source code is written by volunteers and thus is a free and open source project for anyone to use. It is supported by a huge community of followers and also has commercial support, notably CodeWeavers who have helped develop WINE into a solid compatibility layer designed to help non-Windows users install and run Windows applications natively. [5] WINE justifies its existence by saying that it is for people who want to help diversify the operating system market by using it to help provide an alternative to Windows when inevitably Microsoft either ceases operations or implements business practices which do not cater to their customers. It also states that diversifying operating systems leaves less room for viruses to creep into them. They support this claim by quoting systems analysts who said recently that a new worm may be developed and if the OS population is not diversified soon, it could be a great threat to national security. The WINE developers further explain that the dependence on Microsoft is not on the operating system itself, rather the vast number of exclusive applications that are written for the Windows platform because of Microsoft’s monopoly. WINE provides a way to bridge the gap and the normally obligatory paywall to switching out of the Windows ecosystem while still being able to run Windows apps through WINE. Many other companies support and fork off the WINE project in order to have their own implementation. [8]

An example of this is Proton, a compatibility layer based on the WINE framework developed by Valve for their game program called Steam. Proton is the successor to Steam Play and a complete rework of the previous system in which Proton came onto the scene in 2018. It is included in the Linux Client for Steam by default and officially supports over 100 games out of the box, though the number of compatible games is growing monthly. Proton can be force-enabled individually for Windows Steam Games that are not on the official compatibility list, however the actual compatibility of these games with Linux may vary. Unlike WINE, Proton comes with extra features like the compatibility layer DXVK, an implementation that translates Direct 3D 10 and 11 calls into the Vulkan API. Before Proton, WINE was the only way to get apps from Windows working on Linux. Now, Proton helps ditch the extra installation of WINE for games, as long as your favorite Windows games are at least unofficially compatible with Proton. [9]

In the gaming world, a lot of the more interesting games came out in the mid to late 90s and around that time there were three companies providing graphics accelerator cards, commonly referred today as graphics cards. ATI (Later AMD), NVIDIA and 3dfx were the “big three” in graphics cards. 3dfx offered a graphics card option that was very competitive in their Voodoo line of cards. Their cards had good performance, and their Glide API (Based upon OpenGL) was initially superior to their competitors, which used Direct3D, in terms of the simplicity to code for the graphics API. By 2000, developers started to use Direct3D a lot more, after the API was improved, forcing gamers to consider the ATI and NVIDIA cards because 3dfx’s latest Voodoo 3 cards did not support the Direct3D API. 3dfx Interactive filed for bankruptcy when 3dfx’s investors backed out due to poor R&D business decisions by the company in late 2000. That same year, NVIDIA started to absorb 3dfx Interactive for their SLI technology and intellectual property after 3dfx agreed and settled to a 4-year merger with NVIDIA. The source for the Glide API drivers were intentionally leaked for open source in 2003, allowing PC enthusiasts to make their own updated versions of the API to continue support after 3dfx stopped support for the Glide API under NVIDIA’s direction in 2001 and then went defunct in 2002. [10] Two of these API wrappers, based on the original leaked source code, still exist today, dgVoodoo (now called dgVoodoo2) developed privately by Dege and nGlide developed by Zeus Software.

dgVoodoo and its successor dgVoodoo2 are developed by Dege on his website Dege’s stuffs for the express purpose of substituting the Glide API (versions 2.11, 2.45, 3.1 and 3.1 Napalm) in games that only support Glide. It also supports the translation of older Direct X versions up to version 9. It is a drop-in replacement for both APIs and supports Direct X 11 as the output calls of the wrapper. Installation is simple and straightforward, usually achieved by dropping the appropriate DirectX or Glide contents into the main game directory for the game you wish to use dgVoodoo on. The included configuration program can be used to enhance your experience with your game by forcing higher resolutions and other quality of life features not available in some older titles such as anti-aliasing. It is compatible with Windows 7, 8.1 and 10 and works without the use of a 3dfx card. [11] nGlide, developed by Zeus software, is similar to dgVoodoo; however, it only supports translation of the Glide API (versions 2.11, 2.45, and 3.1). The output format for this wrapper is Direct X 9 and Vulkan 1.0 and works without a 3dfx card as well. It is compatible with Windows XP, 7, 8/8.1, and 10. [12]

There are even compatibility layers that have been purposely developed to run apps of different mobile operating systems. Jeremy Andrus, et. al. presented Cider, a compatibility layer to translate the system calls of iOS apps into Android calls on an Android mobile device. This presentation was made during the 19th International Conference on Architecture Support for Programming Languages and Operating Systems in 2014. The abstract states that the compatibility of iOS apps on Android by the Cider compatibility layer is accomplished by a combination of many binary compatibility techniques. Two of them are new approaches called compile time code adaption and diplomatic functions. According to their presentation, compile time code adaption allows unmodified foreign system source code to be used in the host kernel, thereby reducing the implementation efforts for the multiple binary interfaces formerly needed for executing both Android and iOS apps. They also explain that the diplomatic functions approach gives per-thread “personas” which allow foreign apps to use local libraries to access prerequisite proprietary software and hardware interfaces needed for the iOS apps to run. The prototype of Cider presented with this periodical, demonstrated a small performance overhead in Android and was successfully tested on a Google Nexus Tablet running the latest version of Android. It is unclear in the article if the prototype was ever released. [13]

This topic will probably never die out. It is still a viable approach to providing compatibility to older applications. However, the topic is threatened by the growing popularity of software and hardware emulation, especially in the retro gaming scene. A lot of the applications for compatibility layers that have been explored are in the desktop markets and my research implies that there will eventually be movement into the mobile market between Android and iOS cross-compatibility via compatibility layer. Cider was the tip of that exploration iceberg into what could be happening in the next 10 years. If quantum computers become mainstream or even if a 128-bit processor comes out in the consumer market, it could spell the end of the majority of 32-bit compatible applications or even binary-based computer software as a whole (in the case of mainstream quantum computing). In such a case, software compatibility layers need to be written that either translate 32-bit applications into their 128-bit counterparts or in the case of quantum computers, translate any binary computer calls into their equivalents from the realm of a quantum processor. Inevitably, all of these existing layers will endeavor to evolve past the current limitations as hardware and system software evolves. Programmers could find a breakthrough that makes the performance overhead for compatibility layers for PC apps less so than they are now, especially in the WINE project. If such a breakthrough is possible it could spur a renaissance in compatibility layers being the default choice for retro gamers, businesses and consumers everywhere.

Demand for compatibility layers to enable compatibility with retro games and old programs will continue to increase as the age of some older programs with Windows 10 compatibility get closer to incompatibility in the coming years due to Microsoft’s Windows update model. With the known current state of the Microsoft platform, Linux is becoming a more palatable option for a lot of former Windows users and also current Windows users who are becoming increasingly dissatisfied with Microsoft. Thus, I predict WINE will get a lot more attention along with other compatibility layers such as Proton and DXVK. Before this research, I did not know if compatibility between iOS and Android apps on mobile devices was possible given that, from my education on the subject in CIS 255 and 355, most mobile applications are born from Web-based programming languages that are notorious for being non-strict and generally unreliable. The Cider Project is proof that binary compatibility with only a small performance overhead is possible. The new approaches from the Cider project to enable mobile compatibility also were intriguing and would be most likely applicable to other desktop compatibility layers and wrappers. Speaking of wrappers, that area of compatibility layers is going strong with a lot of people getting back into games of the 90s era that still need the Glide API to run smoothly. Both nGlide and dgVoodoo are being actively developed to remain current with the Windows Operating System. At the moment, though, dgVoodoo has better compatibility with both early Direct X and Glide APIs as opposed to nGlide which supports less Glide versions and no early Direct X versions at the moment. However, nGlide may be a compelling option for those using Windows XP for their Voodoo games. It may be worth it to just have a real Voodoo card with the original API if that is the case, though. nGlide has support for advanced features not available on the original cards like resolution forcing, but it appears that dgVoodoo has more options. Most gamers, including myself, tend to support dgVoodoo, although nGlide does have a small cult following as well. Overall it is highly likely that this research shows that there is more to compatibility layers than meets the eye. The applications for compatibility layers as system software continues to expand and shows no sign of stopping. It would be interesting to see if software compatibility layers continue to be developed and improved or if another method would be developed to allow old software to remain compatible with modern system software. Someday current compatibility layers may help spawn ideas for creating so-called universal and intuitive compatibility layers which, as the name implies, would be universally accepting of software from different platforms and eras. Alternatively, operating systems could be inherently designed to have that same universal and intuitive compatibility in mind. These goals for compatibility layers will help to extend compatibility of software beyond the boundaries of current hardware and software limitations and allow for these older programs to not fade away into cyberspace.

References

[1] Emulation General Wiki. (2019). *Compatibility layers - Emulation General Wiki*. [online] Available at: http://emulation.gametechwiki.com/index.php/Compatibility\_layers [Accessed 10 Oct. 2019].

[2] M. Rosenblum, “The Reincarnation of Virtual Machines,” *Queue*, vol. 2, no. 5, pp. 34–40, Jan. 2004.

[3] T. Meyer, “Understanding Shims,” *Microsoft Docs*, 11-Sep-2012. [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-7/dd837644(v=ws.10)?redirectedfrom=MSDN. [Accessed: 14-Oct-2019].

[4] P. Rein, R. Hirschfeld, S. Lehmann, and J. Lincke, “Compatibility layers for interface mediation at run-time,” *Companion Proceedings of the 15th International Conference on Modularity - MODULARITY Companion 2016*, pp. 113–118, 2016.

[5] Q. Radich and M. Satran, “Application Compatibility Toolkit (ACT) - Windows applications,” *Windows applications | Microsoft Docs*, 30-May-2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/win7appqual/application-compatibility-toolkit--act-. [Accessed: 14-Oct-2019].

[6] Emulation General Wiki. (2019). *WoW - Emulation General Wiki*. [online] Available at: http://emulation.gametechwiki.com/index.php/WoW [Accessed 10 Oct. 2019].

[7] WineHQ. (2019). *WineHQ - About Wine*. [online] Available at: https://www.winehq.org/about [Accessed 10 Oct. 2019].

[8] Wine HQ Wiki. (2019). *Importance Of Wine - WineHQ Wiki*. [online] Available at: https://wiki.winehq.org/Importance\_Of\_Wine [Accessed 10 Oct. 2019].

[9] Emulation General Wiki. (2019). *Proton - Emulation General Wiki*. [online] Available at: http://emulation.gametechwiki.com/index.php/Proton [Accessed 10 Oct. 2019].

[10] E. Campidoglio, "The rise and fall of 3Dfx", *Megakemp.com*, 2009. [Online]. Available: https://megakemp.com/2009/01/27/the-rise-and-fall-of-3dfx/. [Accessed: 15- Oct- 2019].

[11] Dege's stuffs. (2019). *dgVoodoo2*. [online] Available at: http://dege.freeweb.hu/ [Accessed 10 Oct. 2019].

[12] Zeus Software. (2019). *nGlide - Glide wrapper | Zeus Software*. [online] Available at: http://www.zeus-software.com/downloads/nglide [Accessed 10 Oct. 2019].

[13] J. Andrus, A. V. Hof, N. Alduaij, C. Dall, N. Viennot, and J. Nieh, “Cider: Native Execution of iOS Apps on Android,” *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems - ASPLOS 14*, pp. 367–381, 2014.