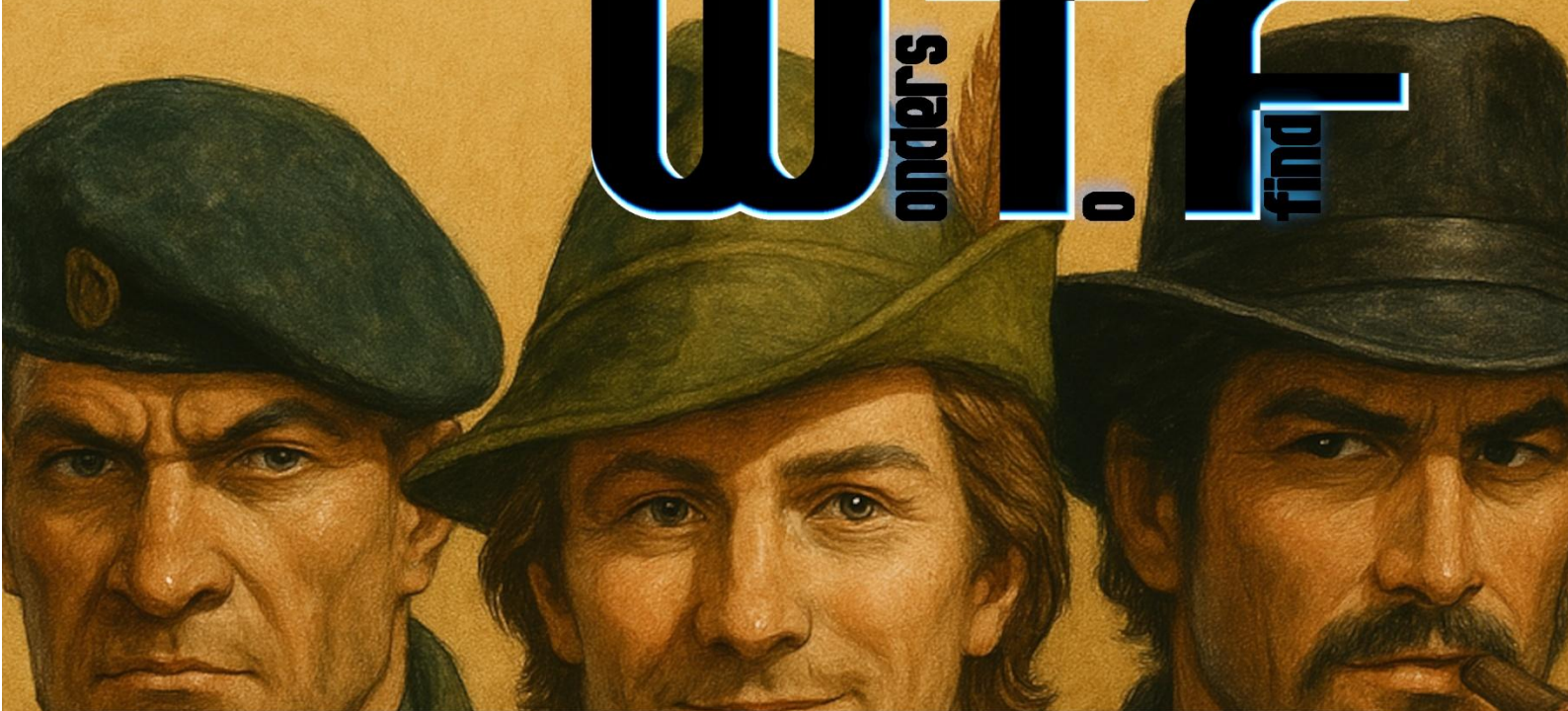


A magazine for the retro generation

NO. 2

Retro

WTF



- ❧ **HIDE AND SEEK THROUGH THE AGES**
- ❧ **DIABLO 2 vs. DIABLO 2 RESURRECTED**
- ❧ **ULTIMA UNDERWORLD - THE
STYGIAN ABYSS**
- ❧ **THE RETRO CODER - GRAPHICS IN
TEXT MODE**
- ❧ **OVER-UNDERCLOCKING TUTORIAL**
- ❧ **RETRO RIG - THE IBM APTIVA 2139**

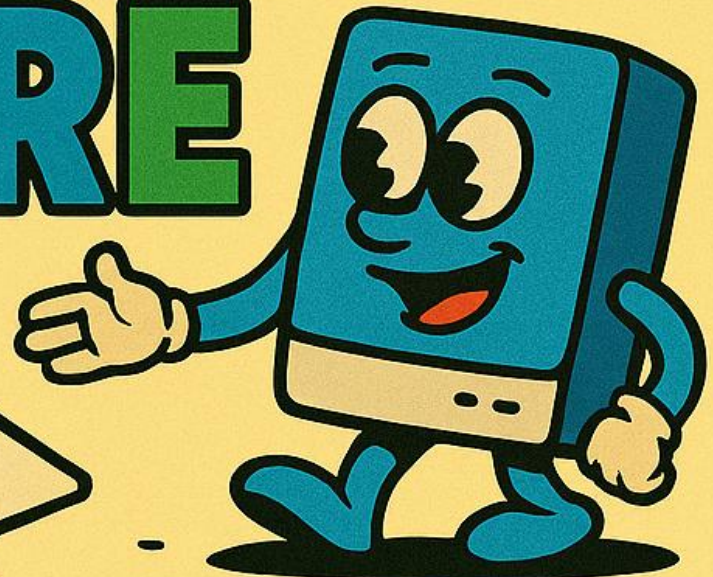
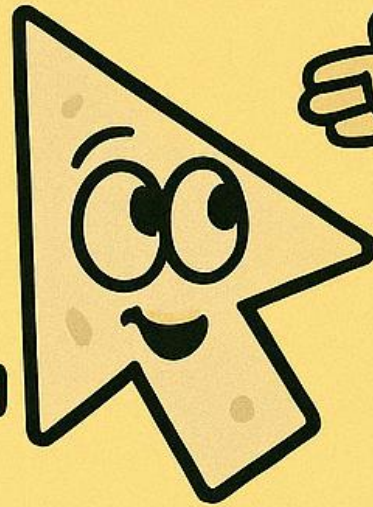
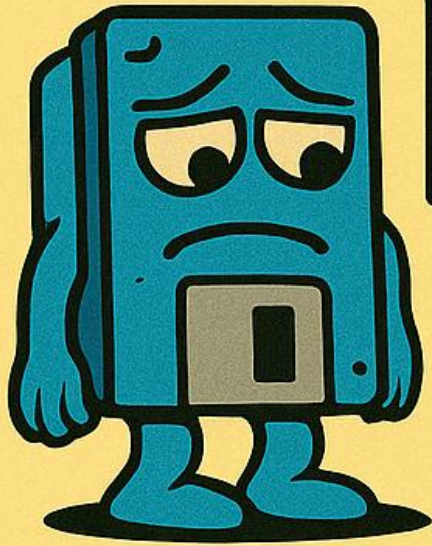


RetroWTF - 002

**IT'S SO BORING
TO PUT UP**



**OUR ADVERTISEMENT
HERE**



**WHAT ABOUT
YOURS?**

Dear Readers ...

Blimey - we've made it to Issue Two! And we haven't even blown a fuse (yet). A heartfelt thank you to all of you who picked up our inaugural issue, squinted at it through bifocals or CRT-induced eye strain, and decided it was worth a second go. We're chuffed to bits to be back, and it's all thanks to your support, kind words, and delightfully pedantic corrections.

Your feedback has been an absolute joy to read - some of it glowing, some of it mildly alarming ("Why haven't you covered the Amstrad CPC 464 in your first issue, but opted for some nasty white piece of brick?!" , "That Compaq is not that heavy," and our personal favourite: "How do you dare to program in C in 2025?"). But every word is appreciated. It's clear that retro computing fans are a passionate, detail-oriented bunch-and we wouldn't have it any other way.

We've now set up a proper home on the web at <https://retrowtf.today/>, where you can download this issue and the previous one (in case your dial-up cut out halfway through the first time). Expect future updates - perhaps even issue number 3, assuming someone will have the time working on it.

As always, we'd love to hear from you. Drop us a line, send in a scan of that obscure manual you found behind the filing cabinet, or regale us with tales of your heroic efforts upgrading from 640K of RAM. Got an idea for an article? Spotted a bug in our retro code? Do get in touch! We're all ears-especially when those ears are tuned to the high-pitched whirr of a 5¼" floppy drive.

Thanks again for joining us on this slightly dusty, wonderfully peculiar journey into the silicon past. Here's to more bytes, more beeps, and a good few laughs along the way.

Cheers,

The Editorial Team @ Retro Wonders To Find

TABLE OF CONTENTS

■	
Welcome	3
BackBreaking News: <i>Oblivion Remake</i>	5
Feature: <i>Fight of the Retro PC Emulators</i>	9
Hide&Seek Through the Ages: <i>Commandos/Robin Hood/Desperados</i>	23
Old Games/New Eyes: <i>THE DIG</i>	33
RetroReview: <i>Ultima Underworld - The Stygian Abyss</i>	36
Old vs. New: <i>Diablo II vs. Diablo II - Resurrected</i>	45
The Hardware Under Your Screen: <i>Overclock/Underclock the CPU</i> ...	52
Show Us Your Rig: <i>The IBM Aptiva 2139</i>	64
RetroCode: <i>C Programming Tutorial - Part 2: Structures</i>	71
RetroCode: <i>Programming the VGA card: Graphics in text mode</i>	80
Retro Society: <i>Hacked</i>	96
Letter to Writers	105



The Elder Scrolls IV: Oblivion Remastered – A Classic Reborn

After years of speculation and mod-driven updates, *The Elder Scrolls IV: Oblivion* has officially returned in the form of a full remastered edition. Developed by **Virtuos Games** in collaboration with Bethesda, the surprise remaster breathes new life into the 2006 RPG classic, blending updated visuals and gameplay mechanics with the rich, immersive world of Cyrodiil that defined a generation of open-world RPGs.

While this article is not a full review, the internet is already teeming with official, unofficial, fans, non-fans and so on reviews, we could not skip this special moment in the history of gaming, considering its retro ramifications, so here the game is presented using our own style without even adding comparison screenshots. I hope, that all of our readers have at least heard of the game, and mostly know how it looks ... well, looked out when it came out. Now imagine, the Remake is better. It's like a colorful Skyrim.

A Technological Leap with Unreal Engine 5

The most immediate change fans will notice is the visual overhaul. Oblivion Remastered has been rebuilt using Unreal Engine 5, providing dramatically improved lighting, environmental detail, character models, and animations. The forests of the Great Forest, the ivory towers of the Imperial City, and the eerie portals of the Shivering Isles now appear with modern fidelity while retaining their original artistic intent.

Now, the bandits have a certain visible smirk on their faces you can see with much better details while they rob you of your hard-stolen coins. This isn't just a cosmetic upgrade, or make-up. It is there to specifically annoy you.

The audio design has been remastered as well, with enhanced ambient sounds, dynamic music blending, and crisper voice lines. Bethesda even added a humorous Easter egg - accessible via a developer command on PC - that lets players interact with a test character using old voice recordings of Todd Howard, adding a nostalgic and quirky charm.

Gameplay Without Losing the Original Feel

One of the biggest criticisms of the original Oblivion was its outdated mechanics, especially in areas like stealth, archery, and character progression. The remaster addresses these issues with careful refinements: stamina management is more intuitive, the stealth system has been modernized to feel more reactive, and blocking and ranged combat are now more responsive and weighty.

Importantly, these changes don't reinvent the wheel—they refine it. The remaster maintains the game's core identity, including its Radiant AI system, skill-based leveling, and the iconic conversation minigame, albeit with quality-of-life tweaks that make them more accessible to today's players.

Community Response: Nostalgia Meets Scrutiny

The response from the gaming community has been largely positive, though not without mixed feelings. On Facebook pages like TheGamer and IGN, longtime fans expressed genuine excitement at returning to Cyrodill in such high quality. Many praised the improved visuals, the smoother mechanics, and the faithfulness to the original tone.

However, some criticism has emerged. A portion of players pointed out bugs that have persisted since the original release, and others questioned whether the remaster justifies its price tag - especially compared to ongoing fan efforts like Skyblivion, a mod project recreating Oblivion inside the Skyrim engine. Still, many acknowledge the convenience of an official, plug-and-play remaster that doesn't require modding knowledge.

Another popular feature is the robust character creator, which has been enhanced to allow for more realistic and expressive faces. Social media has been flooded with screenshots of celebrity lookalikes, hilarious abominations, and creative takes on fantasy heroes, showing just how engaging the customization system has become.

A Shift in Aesthetic: From Vivid Fantasy to Realistic Grit

While many players praised the remaster's visual fidelity and technical advancements, not all reactions have been celebratory. A notable thread of criticism has emerged around the new graphical style—specifically, the loss of the original game's vibrant, almost storybook-like color palette. Fans of the 2006 version fondly remember Oblivion for its lush green hills, glowing magical effects, and warm, saturated tones that gave the world of Cyrodill a dreamlike quality. In the remaster, however, the shift to a more photorealistic style powered by Unreal Engine 5 has muted much of that vividness.

The lighting system, while physically accurate, often casts a neutral tone over environments, making regions feel visually uniform. Some players have voiced concerns that this realistic approach strips the game of its distinctive visual identity. Forests that once felt enchanted now look like generic woodland; mystical cities like Chorrol and Anvil, once glowing with color, now blend into the same earthy, washed-out palette as the rest of the world. "It's technically impressive, but everything just looks... the same now," one fan lamented in a Facebook comment, echoing a sentiment seen across multiple discussion threads.

This visual homogenization has led to cries for a toggle or "classic color" mode, allowing players to reintroduce the original's fantasy-like vibrancy without compromising the updated engine's features. While some fans are already experimenting with post-processing tweaks and shader mods to bring the color back, others argue that this change reflects a broader trend in remasters: sacrificing stylization for realism.

Still, even critics of the aesthetic shift acknowledge the overall polish and improvements. The hope among fans is that future patches—or mod support—will reintroduce some of the visual charm that made the original Oblivion feel so magical.

What's Included and Where to Play

The remaster is available now for PC, PlayStation 5, and Xbox Series X/S, and includes all the original expansions: **The Shivering Isles** and **Knights of the Nine**. Players can purchase a standard edition or a Deluxe Edition, which adds a digital artbook, soundtrack, and several in-game bonuses such as new armor and cosmetic content.

For Xbox Game Pass Ultimate subscribers, the game is available at no additional cost, making it an easy pickup for anyone curious about the hype.

And Finally... A Word to Our Influencer Overlords

Last, but by no means least, let's take a moment to address the sudden flood of content that has hit YouTube and TikTok like a Daedric invasion: the influencer "guides." You know the ones those suspiciously energetic thumbnails shouting things like, *"How to Get the BEST Sword in Oblivion at Level 1!"* or *"Beat the Strongest Enemy in the Game Instantly with THIS Trick with your Level 1 character!"* As if the entire point of a sprawling, carefully crafted RPG was to bypass the journey and jump straight to being an overpowered demigod.

To all the content creators out there breathlessly milking the algorithm with titles like *"Break Oblivion in 10 Minutes"*, may we kindly suggest... touching some Nirnroot and chilling out?

This isn't a battle royale. Oblivion was never designed to be "speedrun to glory while skipping every meaningful challenge." It's a slow-burn role-playing experience, filled with discovery, character growth, and yes, actual *investment*. So, when we see videos proclaiming how to cheese the AI, dupe a Daedric artifact before the tutorial ends, or fast-track yourself to godhood before you've even chosen a birthsign... it doesn't feel clever. It feels like you're trying to microwave a roast.

Here's a radical idea: instead of glitch-hopping your way to some mythical +12 sword of game-breaking, how about you actually play the game? Learn how stealth works. Experiment with spellcrafting. Get lost in a cave, die to a mudcrab, and come back stronger. Because if your idea of "engaging content" is to reduce Oblivion - a game built on immersion, exploration, and slow progression - into a checklist of exploits, then perhaps this isn't the fantasy epic for you.

So yes, dear influencer, we're not here for your tutorials on how to break the system. We're here to lose ourselves in it. Save the shortcuts for the games that don't mind being played like spreadsheets. Oblivion deserves better.

Final Thoughts

Oblivion Remastered isn't just a polished throwback—it's a respectful modernization of a landmark RPG. While not without flaws, it successfully reintroduces the magic of Cyrodill to a new generation of players and provides veteran fans with a compelling reason to return. In a gaming landscape filled with remakes and reboots, this one manages to stand tall—thanks to its careful balance between innovation and reverence.

REMULATING THE PAST

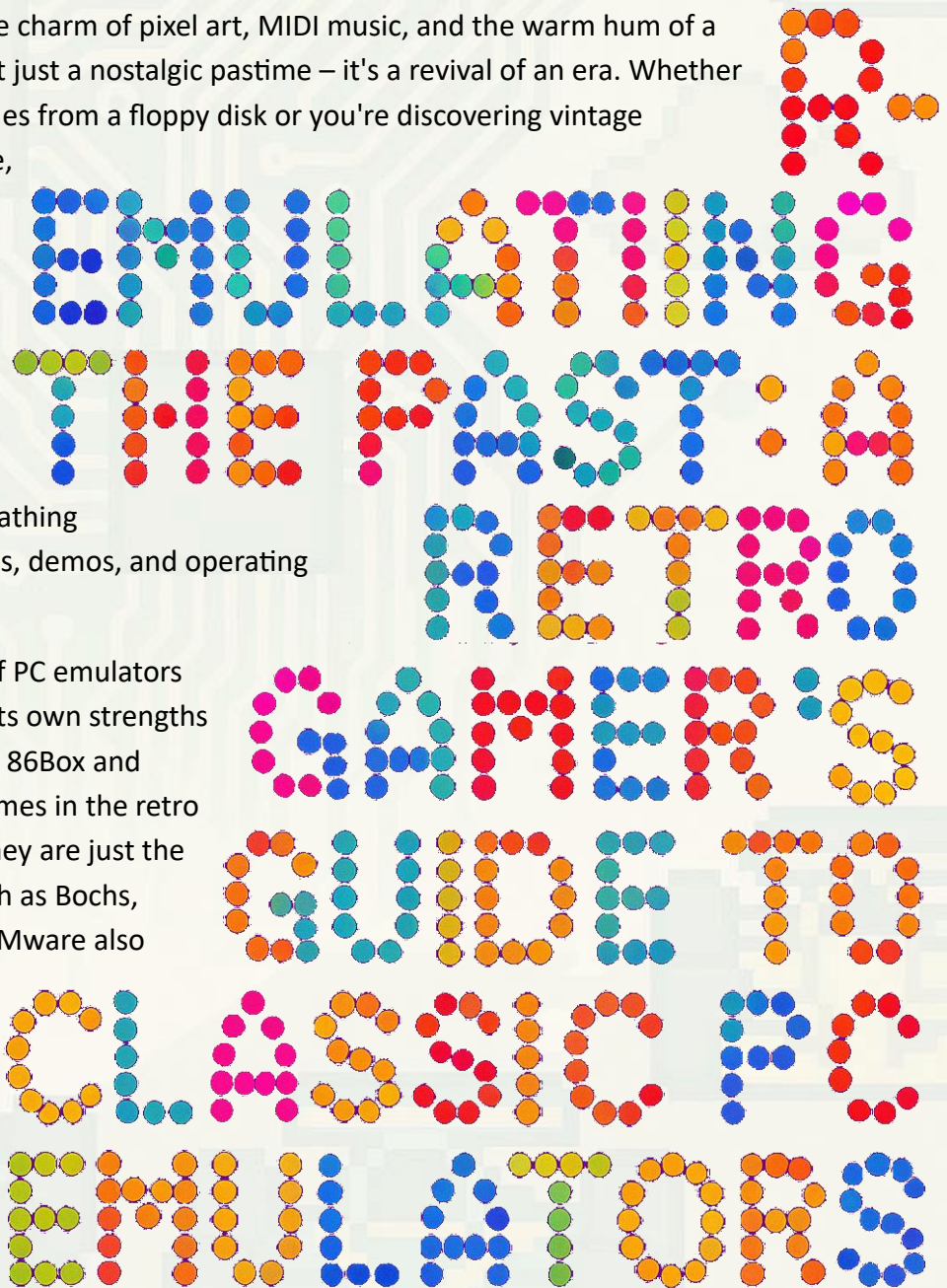
A RETRO GAMER'S GUIDE TO
CLASSIC PC EMULATORS



For anyone steeped in the charm of pixel art, MIDI music, and the warm hum of a CRT, retro computing isn't just a nostalgic pastime – it's a revival of an era. Whether you grew up booting games from a floppy disk or you're discovering vintage software for the first time, the ability to emulate old IBM-compatible PCs is an invaluable asset. Emulators allow us to recreate the computing experiences of the 1980s and 1990s with remarkable accuracy, breathing new life into classic games, demos, and operating systems.

Over the years, a range of PC emulators has emerged, each with its own strengths and quirks. Among them, 86Box and PCem are well-known names in the retro computing scene – but they are just the beginning. Emulators such as Bochs, QEMU, VirtualBox, and VMware also bring an interesting flavour to the table, particularly when considered from the perspective of retro gaming.

In this article, we explore these tools with a focus on how well they serve the retro gaming enthusiast. As a side note for all retro fans out there, we chose the hard way. Our daily machine runs a Linux distro, and we decided to relive all our nostalgic outbursts using this beast. So be prepared: at certain points in this article, we'll mention that we had to compile things ourselves. But since not everyone lives on a penguin farm, we also tested the emulators under Windows to see how they behave there.



86Box: THE POWERHOUSE OF RETRO PC EMULATION

86Box is often hailed as the gold standard for detailed PC emulation. Originally forked from PCem, it has since surpassed its predecessor in both accuracy and breadth of hardware support. For the retro gamer seeking an authentic experience – matching original BIOS behaviour, precise ISA/VLB/PCI bus interactions, and even some obscure hardware configurations – 86Box delivers.

It's particularly invaluable for running early PC games that depend on exact timing, rare graphics chipsets, or proprietary sound cards. Though it demands significant CPU resources and setup time, the payoff is unparalleled fidelity to the original hardware.

Installation Guide for 86Box

Download 86Box: Visit the official 86Box website to download the latest stable version. Choose the version suitable for your operating system (Windows/Linux).

Install the Emulator: Head to <https://86box.net>, and for Windows, download the ZIP file and extract it to a folder of your choice. For Linux, precompiled Appliance binaries are sometimes available for easier access.

Obtain BIOS Files: 86Box requires authentic BIOS files for emulation. These can be legally sourced from old PC hardware or trusted third-party repositories. Follow the instructions on the 86Box website to locate and place the BIOS files in the appropriate folder (usually named roms, located in the same directory as the executable). You may need to have git installed if you want to stay up to date with the latest ROM developments.

Setup: On first launch, configure your emulated PC by selecting the CPU, motherboard, and peripherals. You'll need to tailor the configuration to suit the games or software you intend to run.

Install the OS: Depending on your use case, you'll likely need to install an operating system – just as on a real machine. So, dig out those dusty DOS diskettes (or download a version from your favourite online archive) and get ready to roll.

What We Like About 86Box

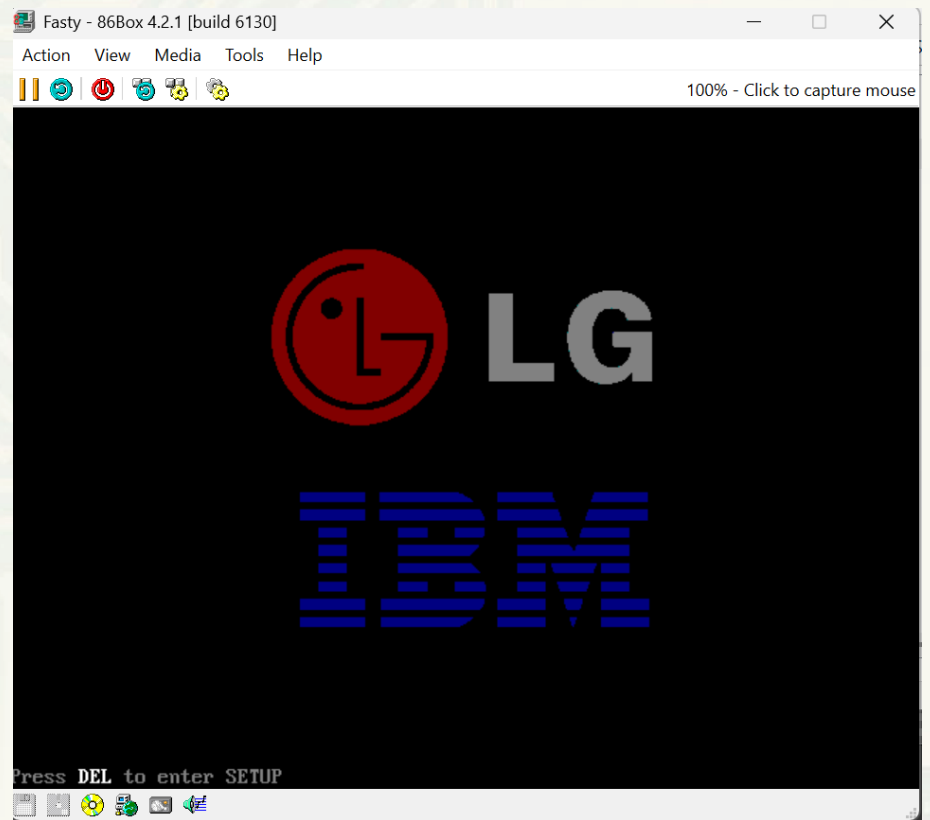
One standout feature is its extensive hardware emulation library. Unlike many emulators that offer only a limited set of parts, 86Box recreates the hardware landscape of the 1980s and 1990s in painstaking detail. Whether you're emulating an 8088-based XT, a 486DX2, or a late '90s Pentium MMX machine, 86Box handles it – and does it well.

Even more impressively, it automatically selects compatible combinations of motherboards, CPUs, chipsets, and peripherals. This means you don't need to be a vintage hardware expert to build a system that makes sense for the era. Support for a broad array of sound cards

(Sound Blaster, Gravis Ultrasound, etc.), network adapters, and SCSI controllers rounds out an enthusiast's dream.

What Could Be Better

The main limitation is its "one machine at a time" workflow. While you can technically run multiple instances, 86Box is fundamentally built to emulate one system per session. If you're juggling multiple retro projects – say, installing Windows 95 on one system while testing a DOS game on another – you'll need to manage configurations manually or set up separate installations.



86Box Emulates a Wide Range of Machines. Some more Exotic than the others...

Bonus: 86Box Manager Makes Life Easier

To ease this pain point, there's a companion tool called 86Box Manager. It provides a graphical front-end for creating, launching, and managing a library of virtual machines – each with its own configuration, disks, and settings. This turns 86Box into something akin to a vintage virtual machine farm, where every setup can be customised and preserved.

However, the Linux version of 86Box Manager, while promising, remains buggy and occasionally unstable. As it's a .NET application ported to Linux, some quirks remain. For now, Linux users may be better off configuring things manually or sticking with the core emulator directly – at least until the GUI becomes more robust.

PCem: THE LEGACY OF ACCURATE PC EMULATION

PCem, once the gold standard of cycle-accurate PC emulation, still holds its own, especially for late DOS or early Windows 95-era gaming. Though no longer under active development, it supports CPUs up to the Pentium MMX and includes 3Dfx Voodoo support – essential for mid-to-late '90s gaming.

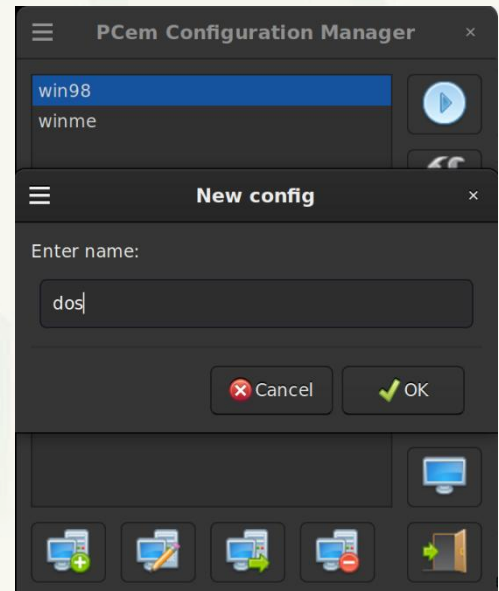
It's also lighter on system resources than 86Box and can run many classic titles more smoothly, making it a pragmatic option for those who value performance without needing exhaustive hardware accuracy.

Installation Guide for PCem

Download PCem: Visit <https://pcem-emulator.co.uk> to download the latest version. **Install the Emulator:** Extract the ZIP archive to a folder of your choice. PCem doesn't use a traditional installer, though you may need to install DirectX manually on Windows. Recent Linux users must compile it from source.

Obtain BIOS Files: Like 86Box, PCem requires BIOS files. These can be sourced from old hardware or located via retro computing communities online.

Configuration: After launching PCem, select your desired system configuration, install an OS, and get ready for some trial-and-error fun – particularly if you choose an exotic machine from 30 years ago matched with some recent hardware. PCem supports floppy and hard drive images. Mount your game and boot into DOS or Windows to begin. It can't mount directories as CD images, but you can create ISO images from directories using tools on your host OS.



Too bad this Machine Manager was not lifted over to 86Box

What We Like About PCem

PCem's optimised emulation engine makes it feel snappy and responsive, especially on modern systems. This makes it ideal for demanding DOS and early Windows 95/98 games, where smooth performance can make or break the experience.

Its model-based hardware selection also adds structure and historical flavour to system setup. Users can recreate period-accurate builds with a sense of purpose, rather than assembling random parts.

What We Really Don't Like

The biggest downside is that PCem has been discontinued. No more updates, bug fixes, or modern host support. The torch has largely passed to 86Box, which builds on PCem's foundations.

Another issue is the lack of constraints on hardware combinations. It's possible to select setups that would never have worked in the real world – leading to confusing or broken

configurations. Newcomers may struggle without some understanding of vintage PC architecture.



Our test App, UnrealGold Runs Pretty Smooth

Bochs: Accuracy at a Cost

Bochs takes a different route: sacrificing usability/speed for instruction-level accuracy. It's the emulator for OS developers and researchers needing to simulate every instruction and interrupt. For gaming, Bochs is generally believed to be too slow (but stay tuned for a surprise), though academically fascinating to experiment with.

Getting Started with Bochs

Download Bochs: Get the latest version from bochs.sourceforge.io.

Install: Windows offers an installer. Linux requires compilation (be prepared for configure lines like `./configure --enable-voodoo --enable-x86-64 --with-sdl2` and `make`). Hard disk creation often needs command-line tools like `bximage`.

Configuration: Setup means manually editing the **bochsrc** file. Specify CPU, memory, and more. Our handy boxout includes a **bochsrc** snippet for Voodoo magic. Some says it is not ideal for gaming, but Bochs can run simple DOS games if configured carefully and if you figure out how to mount a disk image after the system boots (hint: press the configure button, there is the option for changing CDs).

What We Like About Bochs

Bochs takes a distinct philosophical approach compared to 86Box/PCem. Instead of mimicking hardware chaos, Bochs simplifies: a single, configurable virtual machine. No lengthy selection of motherboards or quirky ISA peripherals – configure a clean, abstracted machine via a text file and run. It even supports 3Dfx! This minimalism is a strength, avoiding real-world incompatibilities that 86Box/PCem expose for authenticity. Want MS-DOS 1.0 or early Linux? No worries about graphics BIOS compatibility with sound chipsets There are just a few ones to consider. Bochs is extremely stable and deterministic, a favourite in academic/security circles for testing low-level components or CPU behaviour. One appreciated feature: clipboard access, making pasting long commands easy.

We were impressed by the Banshee emulation speedwhile testing Unreal Gold. Mouse cursor lag is terrible in the Windows GUI, but the game runs smoothly. A pleasant surprise.

What We Don't Like About Bochs

But simplicity comes at a steep price: user-friendliness is minimal. Bochs requires users to write/maintain **bochsrc** files by hand – a learning curve feeling more like programming than using an emulator. Worse, documentation is often outdated/fragmented, and official GUIs are rudimentary if you can compile them. Tasks taking seconds in 86Box – like mounting a CD-ROM – involve manual text edits, frustrating over time. For newcomers, Bochs feels

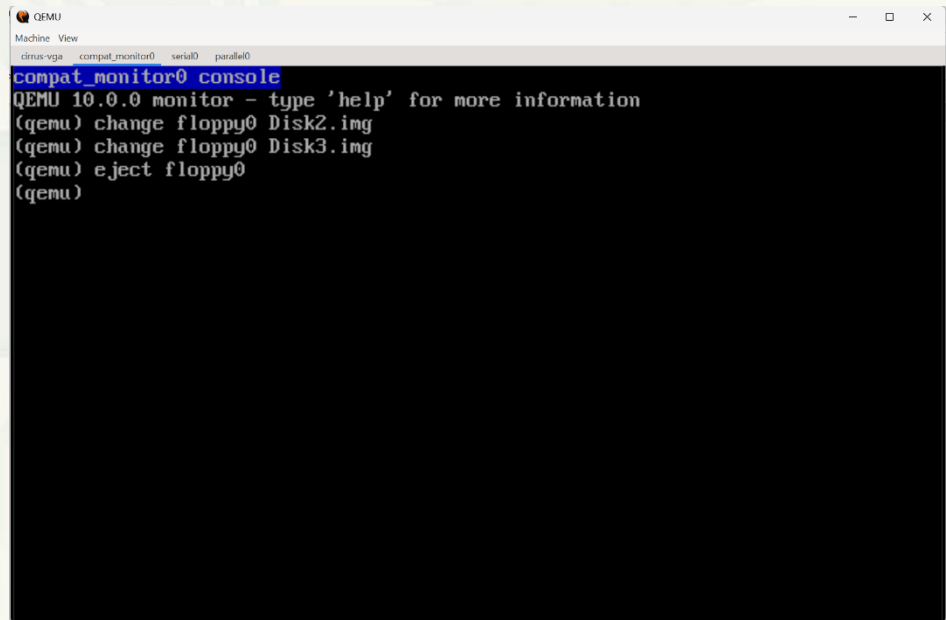


Pretty Pleasantly Surprised with the Emulated Banshee

like a time machine not just for the guest, but the emulator itself. A personal drawback: non-English keyboards can be problematic. Wasting half an hour trying to type **format c:** with a Norwegian layout highlighted this!

QEMU: VERSATILITY FOR THE ADVANCED USER

QEMU occupies a middle ground: versatile, emulating/virtualising systems across multiple architectures. For retro PCs, it supports basic x86 and runs DOS/Win 9x reasonably fast, especially the mouse cursor is blazingly fast in the machine... Yes Bochs, we're looking at you. However, it lacks the fine-tuned hardware needed for accurate graphics/sound in many classic games. Still, for quickly testing old OSes/software in a scriptable, efficient environment, QEMU is handy.



```
QEMU
Machine View
cirrus_vga  compat_monitor0  serial0  parallel0
compat_monitor0 console
QEMU 10.0.0 monitor - type 'help' for more information
(qemu) change floppy0 Disk2.img
(qemu) change floppy0 Disk3.img
(qemu) eject floppy0
(qemu)
```

The Graphical User Interface of qemu is actually a Command Line

Getting Started with QEMU:

Download QEMU: Get the version for your OS from the official QEMU website: <https://www.qemu.org/download/> or from <https://qemu.weilnetz.de/> if you use Windows...

Install: Download the installer on Windows. Linux users can install via package managers (e.g., `sudo apt install qemu`) or choose the painful compilation of 8000 files.

Configuration: QEMU is command-line driven. Configure via specifying CPU, memory, and disk image parameters. When done, install and boot the OS as on a standard machine, and the games are accessible by mounting disk images with lengthy commands.

What We Like About QEMU

QEMU is the Swiss Army knife of emulation. While not focusing specifically on retro PCs, QEMU's superpower is emulating staggering numbers of architectures (x86, PowerPC, ARM, MIPS, SPARC, etc.). It's a cornerstone for developers, researchers, and security analysts across systems. For retro PC fans, QEMU is capable for classic DOS/early Windows, especially mimicking 386/486 machines. It suits building clean environments for legacy software, not cycle-accurate niche chipset emulation.

What We Don't Like About QEMU

Yet for all its power, QEMU isn't beginner-friendly. The steep curve stems from reliance on long command lines with obscure flags/parameters. No built-in GUI exists; third-party front-ends (AQEMU, QEMU Launcher) are often rough or lag feature support. This means getting QEMU to cooperate feels like writing a shell script, even for simple tasks like installing DOS or changing floppies. For many retro fans just wanting to play a game, this lack of user-friendliness is a turnoff, especially when 86Box/PCem offer hardware menus and graphical settings. Also, QEMU lacks viable graphics card emulation (no Voodoo magic!), hindering its retro gaming potential. Cirrus cards weren't built for fast 3D!

VMware: Speed, Polish, and a Corporate Edge

VMware has long been a staple of professional virtualization, and it shows. With fast emulation, polished user interfaces, and excellent integration with host hardware, VMware offers a streamlined experience for running guest operating systems - provided those systems fall within a certain compatibility range. Its ability to leverage the host's resources directly gives it a performance edge that makes even Windows XP feel snappy again. The snapshot feature alone is a huge quality-of-life improvement for testing and tweaking.

Getting Started with VMware

Download VMware: There have been some changes at VMware recently, so finding a site which works for downloading might be troublesome. Techspot has a recent installer, this is the one we used.

Installing: After downloading (from a site which actually works) and installing VMware Workstation (free for personal use), you simply create a new virtual machine by selecting your guest OS and pointing to an installation ISO.

Configuring: The built-in wizard guides you through configuring virtual hardware like CPU, RAM, disk, and network adapters, making it accessible even for those new to virtualization. While installing modern OSes like Windows XP is usually smooth, setting up older systems such as Windows 98 can require additional tweaking due to limited legacy hardware support. Sadly we have failed on installing Windows 98, because we have a too fast machine, however for those who want to dig deeper there is a fix at <https://github.com/JHRobotics/patcher9x>.

What we like about VMware

What we like about VMware is its sheer polish and speed. It's one of the most performant virtualisation environments out there, especially when running operating systems from the Windows NT family onwards. It leverages the host machine's hardware exceptionally well, maybe too well as we have seen in our Windows 98 failure, which translates into fluid guest

experiences and rapid disk I/O. Its user interface is modern, intuitive, and welcoming - the kind of thing that makes you forget you're even using a virtualisation tool. Features like snapshots, shared folders, and drag-and-drop integration blur the line between host and guest in a way that few emulators can rival.

What we do not like about VMWare

But what we don't like is how clear it is that VMware was never designed for retro computing. It exists in the enterprise space, and retro support is incidental at best. Attempting to install Windows 98, for example, often leads to a speed issues (when using newer, faster host machines), poor sound support, and the complete absence of any meaningful graphics acceleration. DOS works, technically, but you're unlikely to get the kind of fidelity a retro enthusiast craves due to DOS runs on a technological power horse, lots of games did not anticipate. Worse still, the product has drifted behind login walls and corporate licensing models, making it harder for casual hobbyists to simply pick up and use.

And a bonus: VirtualBox

While VMware often takes the spotlight, VirtualBox deserves mention as another popular virtualization platform that shares many of the same strengths and weaknesses. Both VMware and VirtualBox excel in delivering excellent performance for the operating systems they support by tapping into the host machine's hardware acceleration, resulting in near-native speeds that far outpace traditional cycle-accurate emulators-especially on modern CPUs. Their polished, user-friendly interfaces make it easy to create and manage virtual machines through simple point-and-click workflows, eliminating the need for manual configuration files or scripting.

However, neither VMware nor VirtualBox were built with true legacy hardware emulation in mind, and they struggle with older systems like Windows 95 or 98, where installation and functionality often break down. Together, they make great general-purpose virtualization tools but fall short when it comes to faithfully recreating vintage PC hardware.



CHOOSING THE RIGHT TOOL FOR THE RETRO JOB

With many emulation/virtualisation tools, choosing can be daunting. Some reproduce 90s hardware quirks accurately (even slow CD access); others offer fast, convenient ways to run old software. Choice depends entirely on your goals – DOS games, early Windows experiments, or exploring esoteric hardware/software. Here's how they stack up:

For Accurate DOS Gaming: 86Box takes the crown. Incredibly accurate emulation of vintage PCs and peripherals. Your 1990s time machine.

For OS Experiments and Surprising Speed: Bochs is ideal for peeking under the hood of Oses. Deterministic and configurable for kernel testing... and provided surprising Unreal Gold speed!

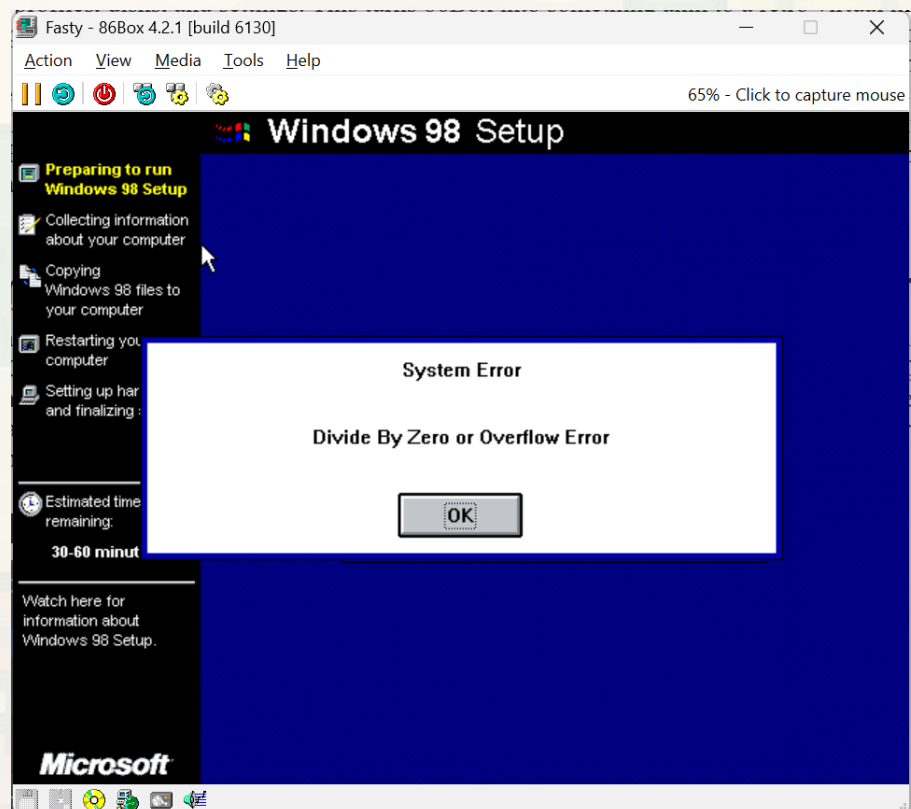
For Hardware Tinkering: PCem still holds up for assembling historically plausible (or implausible!) PC configurations. Experiment with exotic setups.

For Later Windows/Linux Speed: VirtualBox and VMware are unbeatable for Windows 2000, XP, or early Linux. Snappy, intuitive, great for productivity on slightly newer 'retro' systems.

The Versatile Powerhouse: QEMU is the power user's toolkit. Versatile, scriptable, cross-platform, multi-architecture. If you don't mind the command line, it does almost anything.

Ultimately, retro computing emulation isn't just about the software – it's about the hardware you didn't have to source, install, or troubleshoot. Anyone who has configured jumpers on a physical Sound Blaster or hunted Voodoo drivers knows the charm included pain. Emulators

like 86Box let you relive those days – quirks and all – without opening a case or sourcing unreliable vintage parts. Others, like Bochs, strip away the hardware struggle, letting you dive into OS internals or debugging without worrying about chipset compatibility.



Even The Winner Has a Surprise reserved for us from Time to Time

RETRO EMULATOR COMPARISON

Emulator	Best For	Retro hardware coverage	GUI Friendliness	Performance	Notes
86Box	DOS/Win9x gaming & nostalgia	★★★★★	★★★★☆	★★★★☆	Best hardware coverage, great with 86Box Manager
PCem	Hardware experiments	★★★★☆	★★★★☆	★★★★☆	Discontinued, still very usable
Bochs	OS internals, academic use	★★☆☆☆	★★☆☆☆	★★★★☆	Configuration-heavy, very precise, slow mouse
QEMU	Multiplatform and scripting	★★☆☆☆	★★☆☆☆	★★★★☆	Requires lot of command line knowledge
VirtualBox	Later Windows (2000/XP)	★★☆☆☆	★★★★★	★★★★★	Can't run Win98 easily, very user friendly
VMware	Workstation-class retro use	★★☆☆☆	★★★★★	★★★★★	Commercial product, limited retro flexibility, fails to install Win98

What you lose in tactile authenticity, you gain in convenience and experimentation. Build impossible systems, tweak legacy software without fearing capacitor death on your ancient 486. Whether installing Windows 3.1 for the umpteenth time or finishing that DOS RPG, your tool shapes the experience – friction and freedom alike. No perfect way exists, and that's precisely what makes it a rewarding journey.

BEASTS IN BAGS: TOP RETRO RIGS YOU CAN RUN WITH 86Box

As a closing note, we present a list of **high-end PC setups from 1989 to 1997**, capturing what was considered cutting-edge for each year. These weren't budget builds, they're what the most performance-hungry enthusiasts, developers, or early adopters might have had on their desks. From the blazing-fast (for the time!) 486DX systems to early Pentium II workstations with 3D acceleration, these builds reflect the pinnacle of consumer hardware from each era. All configurations are **fully buildable in 86Box**, with period-correct CPUs, motherboards, graphics, sound cards, and more. Whether you want to run Windows 3.1, tinker with DOS games, or fire up early 3D titles, these setups will get you there—faithfully and accurately.

1989 – i386DX/i486 – DataExpert EXP4349 (MR)

This machine will require you to have manually input the type of the hard disk in the BIOS under type 47, so it's better to take note. Also, don't forget the explicit IDE controller, if you work with IDE hard disks.

- **CPU:** Intel i486DX – 50 MHz, the first 486, introduced April 1989
- **Memory:** 4 to 8 MB RAM (very high for the time)
- **Graphics:** VGA 640x480 256 colors (TSENG Labs ET4000AX)
- **Sound:** Creative Labs Sound Blaster (16-bit audio)



TSENG Labs ET4000AX

1994 – The rise of Pentium: Dell Dimension XPS P90

In 1994, the hardware landscape advanced significantly with the release of Intel's Pentium processors into the mainstream, offering improved performance and floating-point capabilities over the earlier 486 chips. For this build we chose one of the classical Dell machines, the Dimension XPS P90. With a fantastic 90Mhz Pentium.

- **CPU:** Intel Pentium 90 MHz
- **Memory:** 16 MB RAM
- **Graphics:** S3 Vision 964 – Diamond Stealth 64
- **Sound:** Sound Blaster AWE32



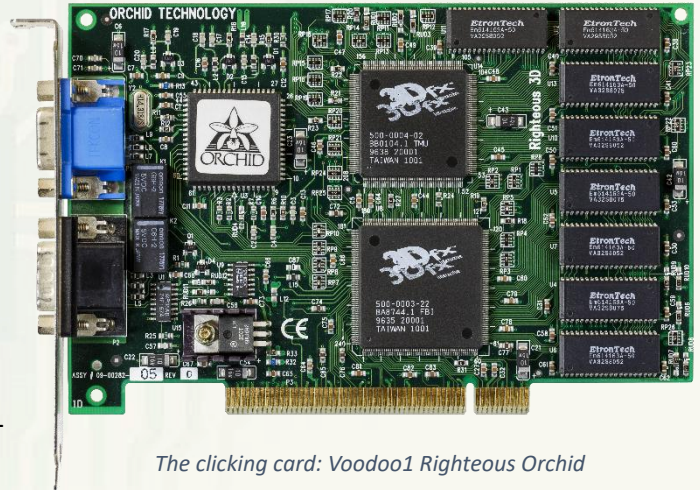
S3 Vision 964 – Diamond Stealth 64

1996 - Voodoo Entered the Building: VIA FP3

1996 was a great year for the computing industry. Intel just released their fastest CPU, the intel Pentium MMX running at a whopping 200Mhz, and a new kid on the block just entered the scene: 3dfx, with their magical 3D card, the Voodoo 1. No more pixelated monster hunting in Quake, using the archaic 320x200, now we can scale up the pixels to 640x480 too (or 800x600 if you have a beefier card with 6MB).

The selection for this category of machines in 86Box is huge, so we have just picked one at random. The machine has a FIC PA-2012 motherboard, with an AGP port too, so you can experiment at your convenience.

- **CPU:** Pentium MMX 200 MHz
- **Memory:** 64 MB RAM
- **Graphics:** S3 Virge (Diamond Stealth 3D-2000) paired with a 3Dfx Voodoo 1
- **Sound:** Sound Blaster AWE64



The clicking card: Voodoo1 Righteous Orchid

At this juncture, we shall refrain from presenting additional configurations beyond the year 1997. While it is entirely possible to emulate later high-end systems, we must admit that 86Box begins to strain the limits of our current hardware when tasked with more demanding CPUs, and we have encountered some instability—particularly with Voodoo3-based graphics emulation. As such, we’ve opted to remain within the bounds of what our machine can handle comfortably and reliably.

Of course, if you are fortunate enough to be equipped with a more powerful system, we wholeheartedly encourage you to explore further. There is great joy to be found in assembling your own ideal late-90s virtual workstation, and we would be most delighted to hear from readers who have crafted their own period-authentic builds. Do feel free to send along your favourite configurations; we may even feature them in a future piece.

As for the imagery accompanying this article, it has been respectfully sourced from community treasures such as DOSDays, Wikipedia, and the invaluable VGA Museum. These are shared here with the utmost admiration for the original archivists. We make no claim of ownership, nor do we derive any financial benefit from their use. They are presented solely for educational and nostalgic appreciation, and no harm is intended.

PLAYING
HIDE AND
SEEK



THROUGH
THE
AGES

“ Whether you crawled through a snowbank to plant dynamite in *Commandos*, fought with three guards in *Robin Hood*, or freed the sheriff in *Desperados* you remember it. Not because of pixels or polygons. But because you **earned** it.

In the late 1990s and early 2000s, before “stealth” became synonymous with AAA sandbox epics or rogue-like indie crawlers, a trio of games emerged that defined and refined a now-niche but once-thriving genre: the **isometric real-time tactics stealth game**. Each game took a different thematic route - *Commandos: Behind Enemy Lines* led us into the heart of World War II, *Desperados: Wanted Dead or Alive* saddled up in the American Wild West, and *Robin Hood: The Legend of Sherwood* lured us into the green-glowing myth of medieval England.

All three were **linked by design DNA** - squad-based gameplay, individual characters with unique abilities, unforgiving enemies, and environments that rewarded patience, creativity, and precision. And yet, each one forged its own identity through atmosphere, mechanics, and tone.

More than twenty years later, these games still spark nostalgia and admiration in equal measure. Here’s a look at how they compare - not just in gameplay, but in legacy.

Robin Hood: The Legend of Sherwood (2002)

While technically the last in the series, historically it can be considered the first, so we think that starting from the medieval times, and advancing toward more modern settings can be in our benefit, to fully comprehend the timeline. Released in 2002 by **Spellbound Entertainment** and published by **Microids**, *Robin Hood: The Legend of Sherwood* is a beautifully crafted real-time tactics, which brought players into the myth and folklore of medieval England - a refreshing change that fused historical romanticism with tight stealth-based gameplay.



Gameplay and Mechanics

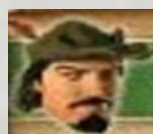
Set in the iconic world of Sherwood Forest, Nottingham Castle, and surrounding medieval villages, the game delivers a meticulously stylized and tactical experience that’s as rich in atmosphere as it is in gameplay depth.

Much like all the other games in this review, the game also centers around **a small team of specialized characters**, each with unique abilities, and places them in sprawling, semi-open maps filled with patrolling enemies, strategic opportunities, and multiple paths to success.

The core gameplay revolves around stealth, timing, and using each character’s strengths in combination. *Robin Hood* leans heavily into **non-lethal gameplay** - promoting knocking out and tying up enemies over outright killing them, or

just simply bribing them with a purse full of gold. Or so they think.

The cast of playable characters is diverse, colorful, and incomplete, because we want to convince you to go and play the game:

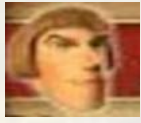


Robin Hood: Agile and quick, he can knock enemies unconscious, climb, use a bow for silent ranged attacks, and blend into crowds. He also leads most missions.



Little John: Strong but slower, he can carry two bodies at once, knock out foes with a punch, and smash down certain doors or gates.

Friar Tuck: A unique utility character who can heal others, carry wine to lure or drug guards, and even persuade people to help the cause.



Will Scarlet: Fast and light on his feet, he's perfect for scouting and picking pockets.

Much the Miller's Son: A nimble thief and rogue with the ability to distract enemies and climb buildings.



Maid Marian: A stealth-focused spy who can disguise herself as a noblewoman to walk through enemy territory freely.

Level Design and Mission Structure

There are **over 30 missions**, set across diverse and beautifully illustrated environments: the lush greenery of **Sherwood Forest**, the muddy streets of **Nottingham**, fortified castles, and even river barges. Each mission has its own narrative context, often connected to a larger campaign to undermine the tyranny of the Sheriff of Nottingham and ultimately rescue King Richard.

Mission objectives range from ambushes and rescues to full-scale sabotage and infiltration, and many maps can be approached in **multiple ways**. You can go in with brute force, using Little John to clear a path, or play ghost-like, knocking out guards and hiding bodies without triggering an alarm.

A day-night cycle adds further variation. Some missions take place under the cover of darkness,



Leaving Three Dead Bodies in the Courtyard is Definitely Not Affecting the other Guards

with limited guard vision, while others occur during the day, requiring more clever use of disguise and distraction.

Controls and Interface

The interface is elegant and medieval, but the author of these lines found the amount of greenery to be a bit of an excess, and very **point-and-click based**, optimized for tactical control. Each character has a context-sensitive menu of actions,

What distinguishes *Robin Hood* is how **fluid and cinematic** the gameplay feels. Characters interact with the environment in dynamic ways - climbing walls, hiding in haystacks, sneaking through windows, and using everyday objects like plates or stools to cause distractions, and also having conversations with in-game characters adds valor to the game.

Robin Hood is very **forgiving and fluid**.

Characters don't die instantly, and enemies are slightly more lenient, making the game more accessible while still retaining tension and it is entirely possible that one main character can have a simultaneous sword fight with three different guards at the same time, while other guards keep patrolling their path happily.

and the game supports **queuing commands** - allowing players to plan out complex sequences (like sneaking, knocking out, and tying up) with a few clicks.



Beautiful Castles in Medieval Britain... What Else can We wish for?

Visually, *Robin Hood* is a **gorgeous 2D isometric game** with hand-painted environments and fluid sprite animations. The attention to detail is remarkable: trees sway gently, villages bustle with life, and interiors are richly textured.

Characters move with lifelike motion, and combat is animated with flair. Watching Robin knock a

One standout feature is the **"tactics" system**, which lets players record a series of movements and commands to be executed later - essentially a primitive macro system. It's incredibly useful for synchronized multi-character actions, like ambushing multiple guards at once.

Graphics and Audio

guard out with a staff, sling him over his shoulder, and dump him into a haystack - all in one seamless flow - never gets old.

The soundtrack is lush and medieval, filled with flutes, lutes, and orchestral strings that evoke the romanticism of Robin Hood's legend. Sound effects are charming - swords clash with a satisfying clang, guards mutter as they patrol, and birdsong fills the

forest air.

Voice acting is light and often tongue-in-cheek, perfectly suiting the game's tone. Robin Hood's quips and the exaggerated English accents reinforce the folk-hero vibe without descending into parody.

AI and Challenge

The enemy AI is relatively intelligent, with focus on relative. Guards react to noises, investigate missing comrades, and sound alarms when suspicious activity is noticed, but the game doesn't lean into unforgiving precision. Players have some leeway, and quick reactions can often save a mission from total failure.

This balance between **challenge and forgiveness** makes the game a tactical title that still respects player creativity.

There are multiple solutions to most problems, and the freedom to explore and experiment is one of its strongest aspects.

On a negative note, however, I have found it a bit disheartening that while I was fighting with three guards in the courtyard the other ones happily went on with their patrol algorithms, undisturbed by the cries for help of their fellow fallen comrades.

Narrative and Presentation

The game blends **historical legend and lighthearted storytelling**, taking the myth of Robin Hood seriously enough to feel grounded but with plenty of playful charm. Cutscenes are brief but well-voiced, often bookending missions with hand-drawn comic-style illustrations that bring the legend to life and providing fun representation of in-game characters.

While not a deeply story-driven game, its episodic structure and recurring characters give it a cohesive feel. Over time, the player truly feels like they are building a rebellion - sabotaging the Sheriff's plans, rescuing allies, and gathering support for King Richard's return.

Desperados: Wanted Dead or Alive (2001)

Our second game in this series was released in 2001 by **Spellbound Entertainment** and published by **Infogrames**. *Desperados: Wanted Dead or Alive* is the second in both senses, both historically, and by release time. The game set in the **American Wild West**, and arguably one of the finest genre successors to the formula introduced by *Commandos* a few years earlier.

Gameplay and Mechanics

At its core, *Desperados* is about orchestrating complex stealth operations using a **team of six diverse characters**, each with unique abilities and personalities. You're given total control of a squad as they undertake missions involving **sabotage, infiltration, rescue, and ambushes**, often against overwhelming odds.

Here's the breakdown of your gang of outlaws:



John Cooper: The central hero - fast, agile, and deadly with his throwing knife. Can silently eliminate enemies, climb obstacles, and dual-wield pistols in shootouts.

Doc McCoy: The team's doctor and sniper. He can knock out enemies with gas, heal allies, throw vials of poison gas, and fire a long-range scoped pistol.



Kate O'Hara: A master of distraction. She can flirt to lure guards, blind enemies with a mirror, or throw perfume to disorient them - but she's unarmed.

Sam Williams: Explosives expert and demolitions man. He can toss dynamite,



plant traps, and use a long-range shotgun. He's loud, messy, and fun.

Pablo Sánchez: A strongman with a bear trap, the ability to carry multiple bodies, and a fondness for brute force. He also commands his pet ferret, "Generál."



Mia Yung: A late-game addition with a blowpipe, healing potions, and a trained monkey named "Mr. Leone" who can steal items and cause chaos.

The gameplay emphasizes **stealth, synchronization, and timing**. You can distract a guard with Kate while sneaking up with Cooper for a quick kill. You can plant dynamite in a patrol path, then pull the trigger at just the right moment. You can clear an entire map without firing a single shot - or cause mayhem if things go loud.

One of the most brilliant features is the **"Quick Action" system** - players can pre-program up to five actions per character and trigger them simultaneously, allowing for **synchronized takedowns** and ambushes. This adds a layer of real-time tactics that's as satisfying as it is strategic.

Level Design and Mission Structure

The game features **25 missions**, set across a beautifully realized Wild West backdrop: dusty frontier towns, desert canyons, speeding trains, military forts, and even Mexican haciendas. Each mission is tightly designed, with intricate patrol routes, interactive objects (e.g., horses, barrels, wells), and multiple approaches.

Each level is **massive in scale** but carefully structured, encouraging scouting and observation. Players must often split up the team and manage parallel objectives while coordinating across large distances, thus seemingly increasing the difficulty of the level.

Environmental interaction is a highlight - knock enemies off ledges, drop crates on their heads, use animals to spook guards, or trigger water towers to cause distractions.

Controls and Interface

The interface is intuitive and functional, especially for a game of its complexity. Each character has a unique command bar with drag-and-drop abilities or hotkeys. Right-clicking brings up radial menus for fast action access.

The game invites improvisation and creative problem-solving in a way that feels dynamic and immersive.

The **Quick Save/Quick Load system** is essential and thankfully generous, given how failure can come fast and unexpectedly.

The **Quick Action (Plan Mode)** is a true standout, allowing players to choreograph simultaneous movements and execute them with one button - a brilliant solution for real-time multitasking.

Graphics and Audio

Even by today's standards, *Desperados* holds up beautifully, especially if you happen to have a smaller screen than today's curved HD monsters. The **isometric 2D environments are hand-drawn with incredible attention to detail** - from bustling saloons and stables to train depots and mines. Dynamic shadows, day-night effects, and bustling environments add realism to every scene.

Animations are smooth and cinematic. Character actions - whether it's Cooper flicking his knife or Kate sashaying to distract a guard - are full of personality and polish.

The audio design is equally rich. Gunfire echoes across the canyon, boots clink on wooden boards, and horses whinny in the distance. The **soundtrack** is a mix of spaghetti-western guitar twangs, saloon piano, and orchestral flair - very much in the style of Ennio Morricone.

Voice acting is top-tier for its time, with snappy dialogue, western slang, and distinct personalities that bring each character to life. There's real chemistry between the cast, with banter and cutscenes that deepen the story.

AI and Challenge

Enemy AI is alert and aggressive, but not unfair. Guards patrol with set routines, investigate disturbances, and respond intelligently to distractions. Sound plays a key role - running, gunfire, or noise-based tools can draw attention or mislead enemies.

You're almost always outnumbered and outgunned. Stealth is not optional - it's the core of the experience. However, the **learning curve is smoother than *Commandos***, and the ability to "pause-plan-execute" means trial and error feels more like learning than punishment.

There's real satisfaction in dissecting each level like a puzzle - clearing zones silently, setting traps, and finally executing your escape or ambush flawlessly.



Thankfully the AI is much cleverer than the Sheriff...

Narrative and Presentation

The story is a surprisingly robust Western adventure. You play as **John Cooper**, a bounty hunter tracking down the mysterious train-robbing bandit known as **El Diablo**. As the story unfolds, Cooper assembles a ragtag team of misfits, each with their own motivations, as they uncover a conspiracy that spans the border between the U.S. and Mexico.

Cutscenes, both hand-drawn and engine-rendered, flesh out the narrative with humor, suspense, and solid pacing. While it doesn't take itself too seriously, the game strikes a great balance between grit and levity.

Legacy and Influence

Desperados: Wanted Dead or Alive was a critical and commercial success in Europe and developed a **strong cult following** globally. It stands as one of the best real-time tactics games of its era and is often considered *the Western-themed counterpart to Commandos*.

Its influence is long-lasting. It spawned sequels (*Desperados 2*, *Helldorado*), and after nearly two decades, the series was revived in 2020 with **Desperados III** by Mimimi Games - a modern masterpiece that paid tribute to everything the original did right.

Commandos: Behind Enemy Lines (1998)

When *Commandos: Behind Enemy Lines* launched in 1998, it carved a niche for itself in the gaming landscape with a mix of real-time tactics, puzzle-like level design, and stealth gameplay set against the gritty backdrop of World War II. Developed by Spanish studio **Pyro Studios** and published by **Eidos Interactive**, it was a bold and cerebral title that defied many trends of the era - a time dominated by fast-paced action shooters and traditional RTS titles.

Gameplay and Mechanics

At its core, *Commandos* isn't about large armies or base-building - it's about a small, elite team of six operatives, each with a highly specialized skill set. Players are given complete control over this unit in a series of meticulously crafted missions, each taking place deep in enemy territory. Several of the missions are reminiscent of novels or Hollywood movies.

Each character has a distinct function in the game:



Green Beret (Jack "Butcher" O'Hara):

Strong, agile, and perfect for climbing walls with knife kills. He can also carry bodies and hide them.



Sniper (Sir Francis T. Woolridge):

Equipped with a sniper rifle and limited ammunition, he's your precision tool for taking out distant threats.



Marine (James "Fins" Blackwood):

Operates in aquatic environments; can

dive, swim, and use a dinghy or scuba gear. He also wields a harpoon gun.



Sapper (Thomas "Fireman" Hancock):

Your explosives expert. Planting dynamite, remote bombs, and laying traps are his specialties.

Driver (Sid Perkins): Handles vehicles and heavy weapons. He is critical in missions involving escape or sabotage.



Spy (René Duchamp): Arguably the most fascinating character. He can disguise himself in enemy uniforms, move freely, and assassinate quietly using a syringe.

The interplay between these characters is the heart of the game. No single operative can complete a mission alone. Success demands synchronized use of their abilities - a sort of real-time chess match with the player as the tactician.

Level Design and Mission Structure

The game contains **20 missions**, each set in detailed and varied environments - from the icy fjords of Norway to sunbaked North African outposts and fortified bunkers in France. These missions are **not procedurally generated**; every enemy patrol, searchlight, minefield, and vehicle is deliberately placed to create a tense, almost puzzle-like experience, and levels toward the end become a real pixel hunting for players with nerves of steel.

What's striking is how Pyro Studios managed to evoke a powerful sense of place and atmosphere

User Interface and

While it may feel dated today, the interface in 1998 was quite intuitive for its genre. Each character had their own icon-based action menu, and players could issue commands via hotkeys or right-click interactions.



I wouldn't stand the if I were You...

despite the game's isometric 2D engine. Soldiers bark orders in German, dogs patrol key areas, sirens scream if you're spotted, and alarm systems can alert the entire map - forcing players to reload and rethink their plan.

There's no room for error. Enemies have **cone-shaped vision fields**, and they'll react to sounds, footprints in the snow, or corpses. A single alert can quickly snowball into failure. The only way to survive is to **observe enemy patterns, coordinate the squad**, and strike with surgical precision.

Controls

That said, the lack of a **mid-mission save system** at launch was one of the game's biggest frustrations.

Missions often took upward of 30–60 minutes to complete, and a single misclick or a patrolling soldier you overlooked could end everything. Later patches added a save feature, which was a relief for many players.

Graphics and Audio

Visually, the game was stunning for its time. The isometric backgrounds were **hand-painted and richly detailed**, evoking the aesthetic of military technical maps crossed with a graphic novel. The visual clarity allowed players to spot important items, distinguish terrain types, and plan escape routes.



... told ya...

The soundtrack was minimal and mostly ambient, punctuated by sound effects - the bark of a dog, the clank of a tank, or the quick mutter of a German soldier. Voice lines for the Commandos were short and often humorous, adding a layer of personality that offset the intense gameplay ("Yes sir!", "Got it!", "Leave it to me!").

AI and Challenge Level

The unforgiving enemy AI is relatively primitive by modern standards, but caused endless hours of

frustration while coining the term "pixel hunting" amongst us, but its predictability worked in the

game's favor. Guards followed strict patrol patterns, and their reactions were consistent - meaning players could plan with confidence. The challenge wasn't about outsmarting evolving AI, but about **solving the level's intricately designed logic**.

Cultural Impact and Legacy

Commandos: Behind Enemy Lines sold over **900,000 copies** in its first year and was particularly popular in Europe, where its cerebral design and historical theme struck a chord with strategy fans, except Germany where due to the graphical representation of various symbols was banned.

Even decades later, the original *Commandos* remains a **cult classic**, often cited as one of the most innovative and challenging strategy games of the late '90s. It also laid the groundwork for its sequels, especially *Commandos 2*, which would refine and expand on its core mechanics.

This gave the game a feel reminiscent of a tactical puzzle box: with enough observation and clever planning, every mission could be beaten without a single shot fired - or in utter chaos, depending on your playstyle.

Commandos: Behind Enemy Lines is a product of another era - a time when games didn't hold your hand, when patience and precision were rewarded, and when a 2D game could still make your palms sweat with tension. Its difficulty curve is steep, its learning curve steeper, but the sense of satisfaction after completing a flawless mission is unmatched.

For those who love meticulous strategy, high-stakes stealth, and the atmosphere of WWII, *Commandos* isn't just a game - it's a rite of passage.

Gameplay vs. Gameplay: Pixel Hunting vs. Creative Flexibility

Commandos was the progenitor. It wasn't just tactical - it was *surgical*. Every mission was a pressure cooker of patrols, gun nests, timing, and enemy sight cones. Mistakes were punished brutally. Guards had hawk-like vision, bodies had to be hidden immediately, and combat was rarely an option. It was a game of **pure logic under pressure** - almost like playing chess at gunpoint.

Each of the six commandos had rigid roles: the Green Beret could climb and stab, the Spy could disguise, the Driver could handle vehicles. There was almost always **one right way** to do things - and the challenge came from discovering and executing it flawlessly.



Commandos was not just about stealth - it was about being smarter than the map.

Robin Hood took the *Commandos* formula and **softened it** - making it more forgiving and more fluid. Characters could knock enemies out and tie them up, and failure wasn't instant death. Maps were more open-ended. Guards were persistent but not omniscient. There was no combat in the traditional sense - only **distraction, timing, and clever evasion**.

Robin and his band were versatile: Marian could pass as a noblewoman, Friar Tuck could bribe or drug people, and Robin himself was the archetypal trickster hero - climbing vines, sneaking past guards, and vanishing into the forest. The **emphasis on non-lethality and mischief** gave it a lighter tone and more options for improvisation.

Robin Hood was still tactical - but it embraced the fantasy of being a legendary outlaw, not a military asset.

Desperados is often seen as the **perfect balance** between the punishing rigidity of *Commandos* and the playful flexibility of *Robin Hood*. It had **intense stealth and patrol mechanics**, but also **robust AI interaction, multiple solutions**, and **tools for synchronizing complex actions** via its Quick Action (Plan Mode) system.

Its cast of six was wildly diverse: Cooper was the knife-throwing hero, McCoy the tranquilizer-slinging sniper, Kate the distraction expert, and so on. Maps were sprawling, reactive, and full of Wild West flair - horses, trains, saloons, mines.

Where *Commandos* had only one or two viable paths, *Desperados* offered a **sandbox of tactics**, inviting players to experiment with traps, timed distractions, and synchronized takedowns.

Desperados was the most cinematic, most expressive, and arguably the most fun of the three - without sacrificing challenge.

Tone and Theme

Commandos: War as chess. Stark, silent, serious. You were behind enemy lines, outnumbered, and likely to die. There was no music during gameplay - just the sound of your own nerves and the bark of discovery.

Robin Hood: Lighthearted but grounded in its myth. The soundtrack hummed with medieval flutes and tambourines, and every mission felt like a storybook caper. You were not just fighting tyranny - you were inspiring the people.

Desperados: Somewhere in between. Stylish, slick, and cinematic. It didn't shy from gunfights, but it wasn't grimdark either. Think *The Good, the Bad and the Clever*. There was a real sense of camaraderie among your outlaws, with character-driven cutscenes and a sense of momentum in the campaign.

Legacy and Influence

Commandos inspired an entire subgenre. It made stealth *intellectual*. Its brutal difficulty and perfectionist design are still studied in game design circles. Though later entries added 3D and lessened the difficulty, none hit the sweet spot like the original.

Robin Hood is often the forgotten gem, loved by fans for its charm and its unique tone. It showed that the genre could be **non-violent, mythological, and even family-friendly** - without losing its tactical core.

Desperados lives on strongest today. It received a stellar sequel in 2020 (*Desperados III*) from Mimimi Games - the modern champions of the real-time tactics genre (*Shadow Tactics*, *Shadow Gambit*). It has a community, modding tools, and enduring replay value. It proved that this genre wasn't just historical - it was *cinematic*.

Each of these games mastered a different aspect of tactical stealth:

- *Commandos* was the **discipline** - harsh, brilliant, and brutally rewarding.
- *Robin Hood* was the **fantasy** - romantic, clever, and mischievously fun.
- *Desperados* was the **style** - fluid, fast, and full of flair.

Together, they represent a **trinity of design ideals** that game developers are still trying to recapture. And for players who love tension, planning, and the thrill of the perfect execution - they are not relics. They are roadmaps.

Thankfully in 2025 all three games are available on Steam, so go on, grab them as fast as your nostalgia kicks in. And of course, Happy Sneaking!

THE DIG



THE DIG

“ We let our Gen-Z contributor to go wild with one of the old classic games, and then kindly asked him to write a review. Does this old classic still hold up to the expectations of the latest generation?

The Dig is a unique entry in the LucasArts adventure game library - less comedic than Monkey Island, more known than Day of the Tentacle. Developed with input from Steven Spielberg (originally written as an episode for his TV-series, Amazing Stories), the game offers a unique mix of classic point-and-click gameplay and an interesting science-fiction story.

You play as Commander Boston Low, an astronaut

leading a team of astronauts sent to divert an asteroid on a collision course with Earth. Things quickly escalate when the asteroid turns out to be hollow, revealing a way into an ancient, alien world. Low is stranded with two other astronauts: one of them being this German fellow called Ludger Brink, who is not only an astronaut, but also an archeologist, and Maggie Robbins, a journalist and linguist. In addition to these three there are also two other characters: Ken Borden, and Cora Miles, but these play

almost no significant role in the story and barely have a reason to exist. In this alien world Low, Brink, and Robbins must discover the secrets and technology of an ancient alien civilization and find a way home.



By modern standards, The Dig seems like a standard point-and-click puzzle game, but the variation of puzzles is comparable to many modern games, even though some of the puzzles make very little

sense and are practically impossible to complete without a guide, the large variety of puzzles and mechanics is still one of the games' greatest strengths.

For a mid-90s title, the visuals of The Dig hold up extremely well when compared to newer games with a similar pixelated artstyle. The immense variety in landscapes and backgrounds is both one of the games greatest strengths, but also create a small but annoying issue that persists throughout the

entire game: The characters walk ridiculously slowly, and when they have to move through dozens, or even hundreds of the large and varied areas the time spent waiting for them to move from one screen to another is almost more ample than the time spent actually doing puzzles and interacting with the game.

Another minor issue with the otherwise splendid graphics is that the contrast between critical items you have to pick up and the background is sometimes non-existent. There is a part where you have to pick up a brown metal plate off the floor, the only problem being that the entire floor on which the brown plate is, is also brown. The game creates a lot of situations where highly important items blend completely into the background, forcing you to painstakingly move your mouse across every single pixel on the screen hoping there is something somewhere.

But ignoring the aforementioned issues, the graphics of *The Dig* are some of the best of

its time, and still hold up surprisingly well to modern standards.



Another sublime component is the audio and sound design, all of the music in the game perfectly complements the scene they are played behind, and further enhance the rich and interesting story. The voice acting is also surprisingly

good, and far surpasses many modern games.

At release, *The Dig* was bombarded with mixed reviews. Some praised its ambition and storytelling, while others criticized its

overly serious tone and sometimes questionably designed puzzles. It was never as beloved as other LucasArts titles, possibly because it strayed so far from their usual formula. However, it has since regained a lot of the popularity and praise it at

first lacked. It still remains an at times strange and mildly frustrating experience, but its interesting and unique narrative, and still amazing graphics complimented by the (mostly) well designed puzzles, make this underrated game a very different experience from other LucasArts entries, and point-and-click games in general.



As per 2025, THE DIG is available on Steam together with a great selection other classical LucasArts games. Go fetch them as long as you can.

Ultima Underworld

THE STYGIAN ABYSS



Ultima Underworld: The Stygian Abyss

“

Because what could be better, on a rainy Tuesday, than slipping beneath the world, where shadows stir and secrets stay - and the Abyss calls your soul to play?”

Released in March 1992 by Blue Sky Productions (later known as Looking Glass Technologies), *Ultima Underworld: The Stygian Abyss* is not merely a game, but a foundational pillar of immersive simulation and 3D RPG design. Published by Origin Systems and set in the renowned *Ultima* universe created by Richard Garriott (Lord British), this title broke conventions and redefined player expectations. It was so far ahead of its time, it felt like it invented new possibilities in game design.

The Background of Development

The development of *Ultima Underworld* was led by Lead Designer Paul Neurath, with Warren Spector (known for his work on *Deus Ex*, *System Shock*, and *Thief*) serving as Producer. Technical masterminds like Doug Church and Dan Schmidt were

instrumental in its creation. The game featured a custom-built, true first-person, texture-mapped 3D graphics engine, a full year before the release of *Doom*. At a time when most RPGs were tile-based, turn-

based, and viewed from a top-down perspective (such as *Wizardry* or *Ultima VI*), *Underworld* offered a fully navigable, real-time 3D world with complete player control over

the camera. It was a true first-person RPG simulator, predating influential titles like *System Shock*, *Thief*, and even *The Elder Scrolls: Arena*.

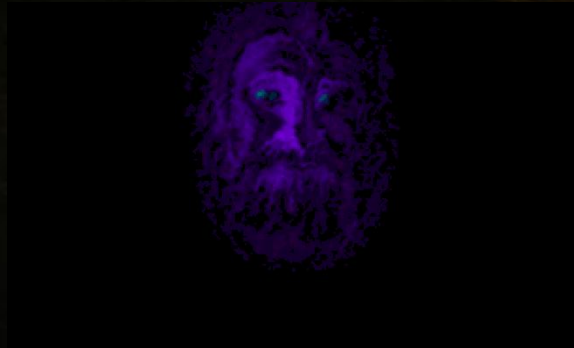
Imagine trying to sell a game concept like

this in the early 90s – “So, it's like *Wizardry*, but you're *inside* the screen, and you can look *up* and *down*! And everything's all... textured!” Revolutionary, indeed.



Setting & Storyline

You assume the role of The Avatar, falsely accused of kidnapping a baron's daughter. Your punishment is to be cast into the Stygian Abyss, an ancient and decaying dungeon built upon the remnants of a failed utopia. Within the Abyss, you encounter lost human outcasts and various peaceful and hostile races, including lizardmen, goblins, and trolls. The Abyss also holds ancient relics and ruins, and is threatened by a growing demonic force that imperils the world above.



Whoever you are, get out of my Dreams

Your mission evolves from mere survival to one of discovery, diplomacy, puzzle-solving, and ultimately confronting a world-ending evil known as The Slasher of Veils. This is far more than a simple dungeon crawl; it's a societal exploration, a mystery, and a narrative masterpiece. You're not just hacking and slashing; you're navigating a subterranean soap opera with potentially apocalyptic consequences!

Key Innovations

The fact that in 1992 the game came with a fully 3D Texture-Mapped World was amazing for the player. The game allowed players to move, jump, look up and down, swim, and even fly. It incorporated the use of light sources, dynamic shadows, and navigable uneven terrain. It was the first game ever to feature a real-time first-person



The First real Door to Almost Nowhere

3D RPG engine. This was mind-blowing at the time. Players could finally experience a dungeon with actual verticality, no longer confined to a flat, top-down grid. Falling down a hole finally felt like *falling* down a hole!

The items in the game could be used in logical and creative ways, such as throwing bones onto pressure plates, mixing reagents, or

utilizing levers. The game also included physics elements like gravity, momentum, and swimming. Puzzles often had multiple solutions, emphasizing player agency. Forget using a key to open a door; in *Underworld*, you might need to stack boxes, use a telekinesis spell, or perhaps bribe a nearby goblin with a moldy cheese to get through.

Players could engage in conversations with NPCs using a parser interface, allowing for full sentences. Negotiation, bartering, and empathy were all viable options. Factions within the game responded dynamically to your behavior, encouraging interaction before combat. You could actually *talk* your way out of a fight with a group of goblins, perhaps by convincing them you had a shinier rock collection elsewhere. Just remember, actions have consequences, and double-crossing a faction might lead to some awkward (and dangerous)

future encounters.

The game featured a rune-based spellcasting system where players discovered and combined syllables to form spells. Experimentation was encouraged, allowing players to potentially create new magical effects through true spell research. Forget memorizing spell slots; here, you were a linguistic wizard, combining runes like "Flam," "Corp,"

and "In" to create fiery, flesh-affecting inward bursts of magical goodness (or something equally chaotic). For players who have not had the chance to own a copy of the original

game book we add a list of all the runes that you can concoct up spells with.

Ultima Underworld was one of the earliest RPGs to effectively use ambient sound cues, dynamic music, and environmental audio. Players could hear a waterfall before

seeing it, footsteps would echo realistically, and audio cues heightened the sense of impending danger. This



Bones ...



More Bones... And a Real Treasure: A Rune-Bag

wasn't just background noise; it was an integral part of the immersion, making the Stygian Abyss feel like a truly living (and often terrifying)

place. Hearing the skittering of a large creature just around the corner with no visual confirmation? Pure, unadulterated dread.

Gameplay Mechanics

Upon start of a new game, players could choose from several fantasy archetypes, including the classical ones, as Fighter, Bard, Mage, and Paladin or the more exotic,

like Shepherd. Various skills were available, such as Appraise, Repair, Casting, Swimming, and Negotiation.

Each skill choice significantly impacted the player's options, encouraging diverse playstyles. Want to be a master haggler who can sweet-talk their way through any situation? Go for it. Prefer to be a hulking warrior who solves problems with a well-placed swing of their sword? Also an option. The game spanned eight massive levels, each with a unique theme. Quests varied widely, ranging from faction diplomacy and language deciphering (including the surprisingly useful ability to learn the Lizardman tongue) to musical

puzzles, artifact recovery, and mystical trials. You might find yourself acting as a diplomat between warring factions one

moment and trying to decipher ancient runes the next. And yes, you literally learn to understand the Lizardmen, which is both useful and slightly bizarre. The inventory system was tactile,

with items represented in a

literal bag and managed through a draggable interface. The game utilized a bartering system instead of traditional money, where players traded items like torches for cheese or gems for armor. Limited inventory space and encumbrance added a survival element to the gameplay. Hoarding every shiny object you find becomes a strategic challenge, and deciding whether that extra loaf of bread is worth leaving behind that slightly-less-rusty breastplate is a genuine dilemma.



Let There Be ... Wendy. RetroWendy

Legacy and Influence

Ultima Underworld is widely considered the spiritual ancestor of numerous influential titles, including *System Shock*, *Thief*, *Deus Ex*, and *The Elder Scrolls*. It also directly inspired elements in games like *BioShock*, *Half-Life*'s immersive environments, *Dishonored*'s systemic gameplay, and *Prey*'s world interactivity.

The game was universally acclaimed by critics upon its release and frequently appeared on "Best Games of All Time" lists for years. As *Game Developer Magazine* aptly put it, "What *Ultima Underworld* did in 1992, most games didn't even dream of until the 2000s". It wasn't just a step forward; it was a giant leap for immersive gaming kind.

The Stygian Abyss is filled with secrets, including hidden rooms, secret factions, and unexpected dialogue trees. The game even allows for a pacifist playthrough, where you can complete the game without engaging in combat. There are also alternate endings based on player choices. Delving into the game reveals secret lore about the Underworld civilization, the Talorids, and the shadowy manipulation of the Guardian. The sheer depth of the world and the various paths you can take make each playthrough a unique experience. Can you truly find *all* the hidden cheese wheels?

Re-Releases and Playing it in 2025

Ultima Underworld, bundled with *Ultima Underworld II: Labyrinth of Worlds*, is available on GOG.com. Community patches and tools like Underworld Exporter allow for playing the game in modern high resolutions. There are also spiritual successors, such as *Arx Fatalis* and *Underworld Ascendant*. And more recently, *Monomyth* seems to gain some traction. So, if you missed out on this gem the first time around, or if your floppy drives are gathering dust, there are legitimate

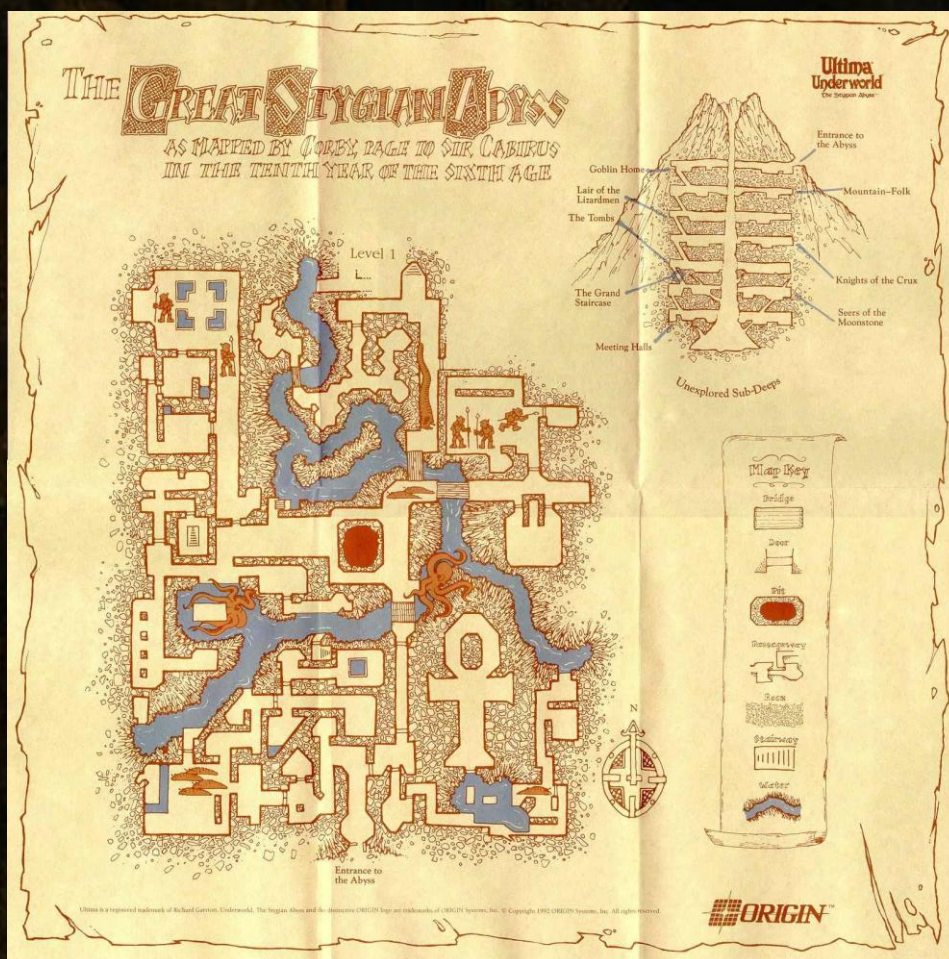
ways to experience this classic today.

A number of dedicated online communities and archives offer rich content related to *Ultima Underworld*, including [The Ultima Codex Underworld Hub](#), [Ultima Dragons Internet Chapter \(UDIC\)](#), and discussions on platforms such as [r/Ultima](#) on Reddit—each preserving and celebrating the legacy of this groundbreaking RPG.

Conclusion

Ultima Underworld remains a landmark in game design, representing a perfect blend of story, systems, and player freedom. It respects player intelligence, encouraging experimentation, immersion, and creativity. It is, in essence, the genesis of the immersive sim genre. Playing *Ultima Underworld* today is more than just retro-gaming; it's like stepping into a living museum of game innovation. It's a reminder that groundbreaking design can come from unexpected places, even a digital abyss filled with goblins and talking lizards.

Ultima Underworld: The Stygian Abyss is not just a game; it's a miracle of software engineering, a design beacon, and a deep, soulful adventure that continues to impress even when compared to modern titles. Whether you are a fan of dungeon crawlers, system tinkerers, narrative lovers, or a historian of game design, this descent into the abyss is an experience you will never forget. Enter the Abyss, and emerge transformed (and perhaps with a newfound appreciation for bartering).



THE EIGHT CIRCLES OF RUNIC MAGIC

THE 1ST CIRCLE

I MA

Create Food (*In Mani Ylem*) Causes a fine bounty of food to appear (permanent spell).

I L

Light (*In Lor*) Illuminates a darkened area (duration spell).

M Φ

Magic Arrow (*Ort Jux*) Fires a magic arrow at your opponent (targeted spell).

B L H

Resist Blows (*Bet In Sanct*) Has the same effect as wearing a suit of head-to-toe armor (duration spell).

H P

Stealth (*Sanct Hur*) Briefly prevents you from making any noise, making it less likely that creatures will notice you (duration spell).

THE 2ND CIRCLE

P K

Cause Fear (*Quas Corp*) May cause an opponent to lose heart and flee (instantaneous spell).

N MA

Detect Monster (*Wis Mani*) Reveals the presence of hidden or unperceived enemies (instantaneous spell).

I B MA

Lesser Heal (*In Bet Mani*) Heals your minor wounds (instantaneous spell).

I Φ

Rune of Warding (*In Jux*) Places an enchantment in an area which will report if anything disturbs it (permanent spell, until disturbed).

R MK

Slow Fall (*Rel Des Por*) Briefly allows you to float in the air like a feather (duration spell).

THE 3RD CIRCLE

B L L

Conceal (*Bet Sanct Lor*) Briefly obscures you, so you might remain unseen (duration spell).

M X

Lightning (*Ort Grav*) Hurls a bolt of arcane energy at your opponent (targeted spell).

P L

Night Vision (*Quas Lor*) Allows you to see without benefit of torch or candle (duration spell).

R L K

Speed (*Rel Tym Por*) Slows down your enemies relative to your speed (duration spell).

H Φ

Strengthen Door (*Sanct Jux*) Spikes a door (permanent spell).

THE 4TH CIRCLE

I MA

Heal (*In Mani*) Heals you of grievous wounds (permanent spell).

P K

Levitate (*Hur Por*) Briefly allows you to rise vertically into the air (duration spell).

L MA

Poison (*Nox Mani*) Poisons your opponent with toxic venom (permanent spell).

M Φ

Remove Trap (*An Jux*) Negates the targeted snare (targeted spell).

H P

Resist Fire (*Sanct Flam*) Briefly grants a partial resistance to damage from flame (duration spell).

THE 5TH CIRCLE

- FT** **Cure Poison** (*An Nox*) Acts as an antidote to any poison (permanent spell).
- KP** **Fireball** (*Por Flam*) Hurls a mighty flaming missile at your opponent (targeted spell).
- XHK** **Missile Protection** (*Grav Sanct Por*) Renders you invulnerable to missiles (duration spell).
- MNA** **Name Enchantment** (*Ort Wis Ylem*) Reveals the true nature of the object on which you cast the spell (permanent spell).
- MA** **Open** (*Ex Ylem*) Unlocks a locked door or chest (permanent spell).

THE 6TH CIRCLE

- AIT** **Daylight** (*Vas In Lor*) Provides bright illumination for extended periods of time (duration spell).
- ARK** **Gate Travel** (*Vas Rel Por*) Allows you to travel instantly to a moonstone (instantaneous spell).
- AIM** **Greater Heal** (*Vas In Mani*) Brings you back to your original vigor (full Vitality) (permanent spell).
- FMK** **Paralyze** (*An Ex Por*) Prevents target from moving (instantaneous spell).
- MKA** **Telekinesis** (*Ort Por Ylem*) Allows you to pick up a single item and use it from a distance (duration spell).

THE 7TH CIRCLE

- IAR** **Ally** (*In Mani Rel*) Causes the ensorcelled being to fight the last enemy he or she saw you attack (permanent spell).
- AFN** **Confusion** (*Vas An Wis*) Causes foes to act as if drunk (instantaneous spell).
- ADK** **Fly** (*Vas Hur Por*) Allows you to fly through the air for a time, and then glide gently to the ground (duration spell).
- AHT** **Invisibility** (*Vas Sanct Lor*) Causes you to become nearly impossible to see (duration spell).
- EPY** **Reveal** (*Ort An Quas*) Reveals hidden objects and concealed exits from current location (instantaneous spell).

THE 8TH CIRCLE

- PD** **Flame Wind** (*Flam Hur*) Casts multiple flaming missiles into the area (instantaneous spell).
- FT** **Freeze Time** (*An Tym*) Stops the flow of time for all but you (duration spell).
- IAH** **Iron Flesh** (*In Vas Sanct*) Greatly increases your resistance to damage (duration spell).
- MKN** **Roaming Sight** (*Ort Por Wis*) Allows you to see the world from a bird's-eye view (duration spell).
- AKA** **Tremor** (*Vas Por Ylem*) Causes the ground to quake and rocks to burst (instantaneous spell).

DIABLO II

VS.

RESURRECTED



“What once was buried now rises again. That’s not just a tagline—it’s a mission statement. When *Diablo II* originally released in 2000, it reshaped the ARPG genre.

More than 20 years later, *Diablo II: Resurrected* aims to reintroduce this classic to a modern audience while respecting its core identity.

When *Diablo II* first stormed onto the scene in 2000, it redefined what an action role-playing game could be. Its blend of dark fantasy atmosphere, addictive loot mechanics, and deep character customization made it a genre-defining title. Over two decades later, *Diablo II: Resurrected* emerges not as a reboot, but as a respectful homage—aiming to modernize the experience while preserving its soul.

But how do these two versions truly compare? Let’s dive into every detail: from graphics and gameplay to modding and multiplayer.

A VISUAL RESURRECTION

From Gothic Sprites to 4K Hellscapes

One of the most immediately striking differences between *Diablo II* and *Diablo II: Resurrected* is the graphical overhaul. The original ran at a modest 640x480 resolution, built on a 2D sprite-based engine. At the time, it was gorgeous - brooding cathedrals, windswept deserts, and monster designs that burned into your memory. But even with the Lord of Destruction expansion, the resolutions were capped at 800x600, and the assets were pre-rendered and static.

THE OLD



THE NEW



Resurrected, however, brings the world of Sanctuary to life in full 3D with support for resolutions up to 4K. Characters, environments, and monsters have all been meticulously rebuilt with modern rendering techniques—complete with dynamic lighting, ambient occlusion, shadows, and even HDR support. The result is stunning without being intrusive. The **original grid-based layouts and hitboxes remain unchanged**, ensuring gameplay feels identical. And for purists? A single button press lets you instantly toggle back to the classic look, offering a nostalgic glimpse at how far the game has come.

Visual Toggle Feature: With a single keystroke (default: G), you can instantly swap between legacy and modern visuals—perfect for appreciating the visual fidelity without forgetting your roots.

GAMEPLAY

Untouched by Time

At its core, *Diablo II: Resurrected* plays exactly like the original. The simulation engine underneath - the one handling combat, loot rolls, pathfinding, and skill behavior - is untouched. Every mechanic, every exploit, every frame of animation is preserved, however it became possible to modify certain aspects of the game, such as Item Drop Spacing to match the new or the old tastes.

You still get:

- **Seven character classes:** Amazon, Sorceress, Necromancer, Paladin, Barbarian, Druid, and Assassin
- Runewords, Horadric Cube recipes, mercenaries, PvP dueling, and boss farming
- A brutal and rewarding **difficulty curve** through Normal, Nightmare, and Hell modes

That said, optional **post-launch content**—such as **Terror Zones** and **Sunder Charms**—spices up endgame content for those seeking new challenges.

THE SOUND OF HELL

Sounds even better with 5.1 extra channels

While the original's audio design by Matt Uelmen, haunting music, ambient effects, and the clang of swords - was praised for its moodiness, it was limited

to stereo output. *Resurrected* remasters that iconic soundscape for 7.1 surround sound, offering enhanced immersion without altering the core compositions. It even includes a legacy mode for those who prefer the original audio mix.

CONTROLLER, SUPPORT, AND ACCESSIBILITY

The controllers work especially good on consoles

With *Resurrected*, *Diablo II* breaks free from its mouse-and-keyboard roots. Full controller support has been added, with an intuitive interface tailored for console play. The game is now available on **PlayStation, Xbox, and Nintendo Switch**, opening the gates of Hell to a broader audience.

Accessibility options have also been expanded, including scalable text, colorblind-friendly UI settings, and an updated control scheme designed to be more comfortable for modern players.

A surprising feature crept in to *Diablo II: Resurrected*. It supports **importing your original Diablo II save files** (as long as they were created as local/offline characters).

The steps for these can be quite tricky, but with a bit of guidance we think it will not represent a big problem.

Firstly, You just have to copy your old save folder over to `c:\Users\<YOURUSER>\Saved Games\Diablo II Resurrected\` and this should sort out all the details. You even can upgrade the characters to the expansion if required.

From this point, just start the game, and either it works out of the box, or



So that's how a 30-year-old Necromancer looks like

not. Since we fell into the second category, we had to do some troubleshooting after the strange new messages, like “Could not join the game” or “Cannot access file”.

If you get strange error messages, like could not join game, a small trick or two for you in order to sort out these annoyances:

1. Make sure the save files (*.d2s) are NOT read-only. Some backup systems set this flag. This fixed our character upgrading, which strangely could not rewrite the files after the upgrade. Makes sense somehow.
2. If step one failed, make sure the file names have a lowercase ending: .d2s not .D2S. It seems somewhere in the game the lowercase ending is hardcoded.
3. When picking to join a game, pick the highest possible level, ie. the one you ended up with 20 years ago. We tried to pick the easiest one for our Nightmare character but we failed.

These tricks have helped the author of these lines to regain control of their level 33 Necromancer to effectively go and hack monsters all around again. Or even better raise skeletons. A lots of skeletons to run around you like crazy.

THE OLD



THE NEW



MULTIPLAYER: FROM LAN PARTIES TO GLOBAL LADDERS

You still can play with your mate in the other room

Back in 2000, *Diablo II* supported TCP/IP multiplayer and open Battle.net play—along with all the exploits, bots, and dupes those systems invited. *Resurrected* opts for modern Battle.net integration, with stricter anti-cheat protections and region-wide ladder seasons.

Gone is local multiplayer via LAN or TCP/IP, which some old-school fans lament. But in its place is a smoother, more secure online experience that supports **cloud saves** and **cross-progression** between platforms (though not cross-play).

MODDING: THEN AND NOW

Because that's how you make your game your own

Modding kept *Diablo II* alive for years after its prime—mods like **Median XL**, **Eastern Sun**, and **Path of Diablo** pushed the engine to its limits.

The original relied on binary hacks and .mpq file editing. In *Resurrected*, Blizzard introduced a **modding framework** that supports override-friendly .csv and data files. While not as open as full source access, it's a major improvement.

Some classic mods won't work without reengineering, but the foundation for modern modding is solid and growing.

FINAL VERDICT: WHICH DIABLO SHOULD YOU PLAY?

This should not be a question now

If you're a **veteran** returning to Sanctuary, *Resurrected* feels like coming home—with a fresh coat of paint, more storage space, and no LAN cables in sight. If you're a **new player**, this is hands down the best way to experience one of the greatest ARPGs ever made.

Most importantly, *Diablo II: Resurrected* succeeds where many remasters fail: it modernizes without sterilizing. Every frame of its haunting, addictive brilliance has been honored.

I think, with these words we can conclude, that unless you have a retro machine, *Resurrected* lives up to the moods of the players, so it's time to jump in it without regrets that you will out something from the original. You won't. Not even your old characters you played with 20 years ago.



SYSTEM REQUIREMENTS

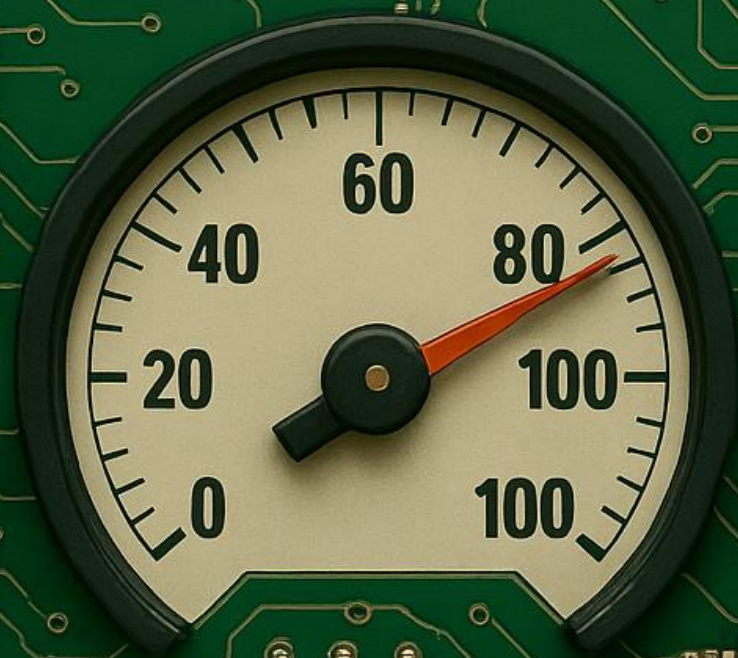
Then vs. Now

When *Diablo II* launched in 2000, its requirements were modest—even by that era's standards. A **Pentium II CPU**, **64 MB of RAM**, and a basic **DirectX-compatible GPU** were enough to descend into Hell.

Fast forward to *Diablo II: Resurrected*, and the bar is much higher. With its 3D graphics and modern rendering pipeline, the remaster requires a **quad-core CPU**, **8 GB of RAM**, and at least a **GTX 660 / Radeon HD 7850 GPU**. An SSD with at least 30GB free is strongly recommended for smoother load times and zone transitions.

While the original ran comfortably on late-90s hardware, *Resurrected* requires a mid-range gaming PC or current-gen console. That said, it's still very accessible by today's standards and runs well even on modest modern setups.

OVERCLOCKING



UNDERCLOCKING

A quick guide to Overclocking and Underclocking a CPU



Because what can be more optimal than running a CPU slower or faster than their intended speed.

Modern (read: from the 90's onwards) central processing units (CPUs) are designed with a careful balance of performance capabilities, energy efficiency considerations, and thermal limitations that dictate their operational boundaries. However, many advanced users and retro enthusiasts seek to go beyond these factory settings, often attempting to modify a CPU's operating parameters to either extract additional performance through overclocking or to minimize power consumption through underclocking. While both practices involve altering the CPU's fundamental behavior they are driven by fundamentally different objectives and carry distinct sets of implications for system stability and longevity. This article provides a comprehensive and detailed exploration of both overclocking and underclocking, delving into not only the practical methods involved but also the underlying electrical and physical principles that govern their effects.

Understanding Clock Speed

In a modern computer system, the clock signal that drives the CPU and other subsystems originates from a **crystal oscillator**, typically a **quartz crystal** soldered onto the motherboard. This oscillator is a piezoelectric component that vibrates at a highly stable frequency when voltage is applied, most commonly producing a base frequency such as **25 MHz** or **100 MHz**. However, this raw signal is not directly used by the CPU; instead, it is fed into a dedicated **clock generator chip**—such as the **IDT 9-series** (Integrated Device Technology), **ICS clock generators**, or other similar timing ICs. This chip often includes a **Phase-Locked Loop (PLL)** circuit, which takes the base crystal frequency and multiplies it to generate the higher frequencies required by the CPU and other high-speed components. For instance, a 100 MHz base clock (commonly known as the **BCLK** on Intel systems) can be multiplied by an internal multiplier—controlled by the CPU or firmware—to achieve operational frequencies in the gigahertz range, such as 3.6 GHz or more.

The **Intel Platform Controller Hub (PCH)** or **AMD Fusion Controller Hub (FCH)**, as part of the motherboard chipset, also plays a role in distributing and synchronizing these clock signals across various system buses like PCIe, SATA, USB, and DRAM. These components ensure that all parts of the system operate in

harmony with precise timing. Importantly, although CPUs like Intel's Core i7 or AMD Ryzen appear to manage their own dynamic frequencies via technologies like **Intel Turbo Boost** or **AMD Precision Boost**, the foundational timing still depends on the external clock source generated by the motherboard's oscillator and clock generator circuitry.

This clock signal lies at the core of a CPU's operation, which is a consistent and rhythmic square wave of voltage: the signal acts as the metronome that dictates the pace at which the processor executes instructions; each rise and fall of the wave marks a clock cycle, a discrete unit of time during which the CPU performs its operations. Clock speed is measured in Hertz (Hz), representing the number of cycles per second, and in modern CPUs, this frequency is typically in the gigahertz (GHz) range, where 1 GHz is equivalent to 1 billion cycles per second.

Within each clock cycle, the CPU can perform a small, discrete unit of work. This work might involve a variety of fundamental operations, such as fetching an instruction from memory, decoding that instruction to understand what needs to be done, executing the instruction (performing the calculation or data manipulation), or writing the result of that operation back to memory. In essence, the

clock cycle provides the timing framework within which these fundamental CPU operations occur.

The clock speed is directly related to the CPU's potential processing capacity. A higher clock

speed means that more clock cycles occur in a given second, and since each cycle allows the CPU to perform a piece of work, a faster clock theoretically translates to the ability to process more instructions and complete more tasks in the same amount of time.

Overclocking: Boosting Performance

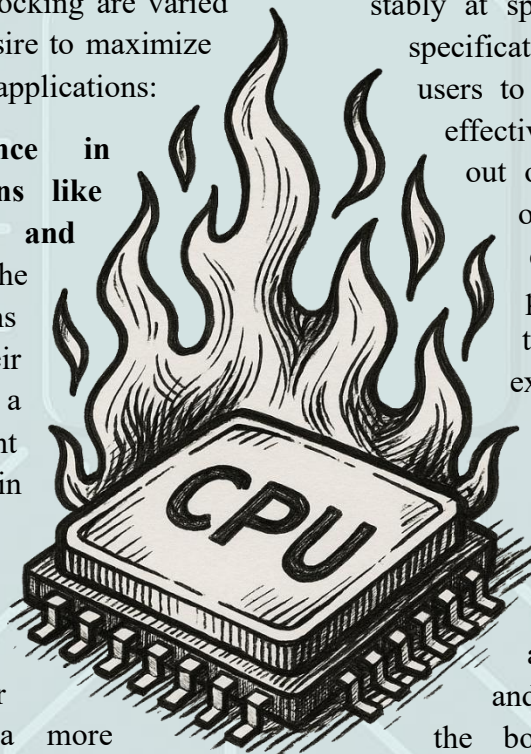
Overclocking is the practice of configuring a CPU to operate at a clock speed that exceeds its factory-specified or rated operating frequency. This deliberate pushing of the CPU beyond its intended limits is done with the goal of achieving higher performance. For example, a processor that is designed to run at a stock speed of 3.0 GHz might be overclocked to run at 3.6 GHz or even higher speeds, depending on the CPU's capabilities and the system's cooling capacity.

The motivations for overclocking are varied and often driven by the desire to maximize performance in demanding applications:

- **Enhanced performance in demanding applications like games, simulations, and rendering:** One of the primary reasons individuals overclock their CPUs is to gain a noticeable improvement in performance in computationally intensive tasks. In video games, overclocking can lead to higher frame rates and smoother gameplay, providing a more immersive and responsive experience. Similarly, in scientific simulations or 3D rendering, overclocking can slightly reduce processing times, allowing for faster completion of complex calculations or the generation of intricate visual content. This pursuit of increased performance is particularly relevant for users who require

maximum processing power for their work or entertainment.

- **Maximizing the potential of hardware, as some CPUs can operate beyond their rated specifications:** Manufacturers often set conservative clock speed ratings for CPUs to ensure stability and reliability across a wide range of operating conditions and system configurations. However, many CPUs possess a degree of inherent performance headroom and can operate stably at speeds exceeding their official specifications. Overclocking allows users to tap into this extra potential, effectively getting more performance out of the hardware they already own. This can be a cost-effective way to boost performance without the need to purchase a new, more expensive CPU.



- **Experimentation and the pursuit of performance by us, technology enthusiasts:** Overclocking is also a popular activity among technology enthusiasts and hobbyists who enjoy pushing the boundaries of hardware and exploring the limits of CPU performance. For these individuals, overclocking is not just about achieving a specific performance gain but also about the challenge and satisfaction of fine-tuning their systems and optimizing performance. It's a way to learn about the intricacies of computer hardware

and to engage in a community of like-minded individuals.

Overclocking is typically achieved by carefully adjusting several key parameters within the computer system's BIOS/UEFI firmware or through specialized software utilities:

- **Base Clock (BCLK):** The base clock is the fundamental timing frequency that serves as the foundation for many subsystems within the computer, including the CPU, memory, and interconnects. It acts as a reference frequency from which other operating speeds are derived. Increasing the BCLK raises the operating frequency of these interconnected components. However, it's crucial to adjust the BCLK cautiously, as it can impact the stability of the entire system.
- **Multiplier:** The CPU multiplier is a factor that is applied to the base clock to determine the final operating frequency of the CPU cores. The CPU frequency is calculated by multiplying the base clock by the CPU multiplier ($\text{CPU Frequency} = \text{BCLK} \times \text{Multiplier}$). For example, if the base clock is 100 MHz and the multiplier is 30, the CPU frequency will be 3.0 GHz. Adjusting the multiplier is a common and relatively straightforward way to overclock the CPU, as it primarily affects the CPU's speed without directly altering the speeds of other system components.
- **CPU Core Voltage (Vcore):** The CPU core voltage (Vcore) is the amount of electrical voltage supplied to the CPU cores. Increasing the Vcore is often necessary when overclocking to provide the CPU with the additional power it needs to operate stably at higher frequencies. However, increasing the Vcore also leads to increased heat generation, which necessitates adequate cooling to prevent damage to the CPU.

Overclocking is generally performed through two primary methods:

- **BIOS/UEFI Interface:** The Basic Input/Output System (BIOS) or its modern replacement, the Unified Extensible Firmware Interface (UEFI), is the firmware embedded on the motherboard that controls the computer's basic hardware functions. The BIOS/UEFI interface provides access to a range of settings, including those related to CPU clock speeds and voltages, allowing users to manually adjust these parameters to achieve their desired overclock. Overclocking through the BIOS/UEFI offers a high degree of control but requires a good understanding of the system's hardware and settings.
- **Software Utilities:** Several software utilities are specifically designed to facilitate overclocking within the operating system environment. These utilities, such as Intel XTU (Extreme Tuning Utility) and AMD Ryzen Master, provide a user-friendly interface for monitoring system parameters and making real-time adjustments to CPU frequencies and voltages for modern processors that have unlocked these features.

But I hear you, dear reader: In my retro machine I have an old motherboard, with basic BIOS ... and I still want to overclock. You're raising a very interesting and historically significant point about overclocking, because well, in the end this is a magazine for retro computing! It's true that early overclocking often involved methods beyond simple BIOS adjustments. Here's a more detailed explanation of how overclocking was achieved directly on the motherboard, especially in the days when BIOS options were limited or non-existent for this purpose.

Underclocking: Enhancing Efficiency

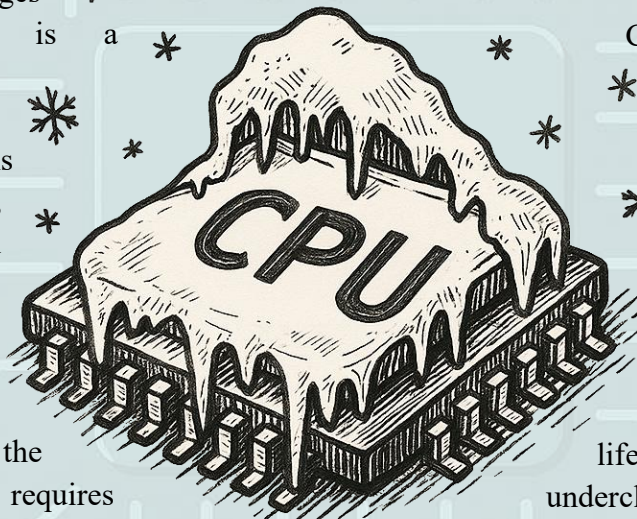
Underclocking is the opposite of overclocking; it is the process of deliberately reducing the CPU's operating frequency below its default or factory-specified value. Instead of pursuing maximum performance, underclocking prioritizes energy efficiency, reduced heat generation, and increased system longevity, to not to mention that games, such as some of the Ultima series, whose running speed was directly affected by the speed of the CPU they were running on could be enjoyed in their natural environment, even if one had a newer computer they were programmed on.

The benefits of underclocking include:

- **Lower power consumption:** One of the most significant advantages of underclocking is a reduction in the CPU's power consumption. As discussed earlier, power consumption is directly related to the CPU's operating frequency and voltage. By lowering the frequency, the CPU requires less power to operate, leading to energy savings. This is particularly important in battery-powered devices like laptops, where reducing power consumption can significantly extend battery life.
- **Reduced heat output:** Lower power consumption directly translates to reduced heat generation. The less power the CPU consumes, the less heat it dissipates. This can be beneficial in various situations, such as when building a quiet computer system or when operating in environments with limited cooling capacity. Reduced heat

output can also contribute to increased system stability and longevity by minimizing thermal stress on components.

- **Extended battery life (especially for laptops):** In portable devices like laptops, underclocking is a crucial technique for maximizing battery life. By reducing the CPU's power consumption, underclocking allows the battery to last significantly longer, enabling users to work or play for extended periods without needing to recharge. This is particularly valuable for users who rely on their laptops for mobile productivity or entertainment.



- **Increased longevity and stability:** Operating a CPU at lower frequencies and voltages can contribute to increased system longevity and stability. Reduced heat and power consumption lessen the stress on the CPU and other components, potentially extending their lifespan. Additionally, underclocking can improve system stability, especially in situations where the system is prone to overheating or instability at its default settings.

Underclocking is closely related to another technique called undervolting, which involves reducing the CPU's core voltage. Since power consumption is quadratically related to voltage, even small reductions in Vcore can lead to substantial power savings and reduced heat generation. In many cases, underclocking is combined with undervolting to achieve optimal energy efficiency and thermal performance.

Early Motherboard-Level Overclocking Techniques

In the early days of PC overclocking, before BIOS interfaces became sophisticated and user-friendly, enthusiasts had to rely on hands-on, hardware-based modifications to push their systems beyond factory-rated specifications. Unlike today's software-assisted overclocking tools, early methods required a deep familiarity with motherboard schematics, physical dexterity, and a healthy dose of experimentation. This often meant physically altering the motherboard itself, with tools like tweezers, soldering irons, and a steady hand. Let's take a closer look at some of the most common techniques used by early overclockers:

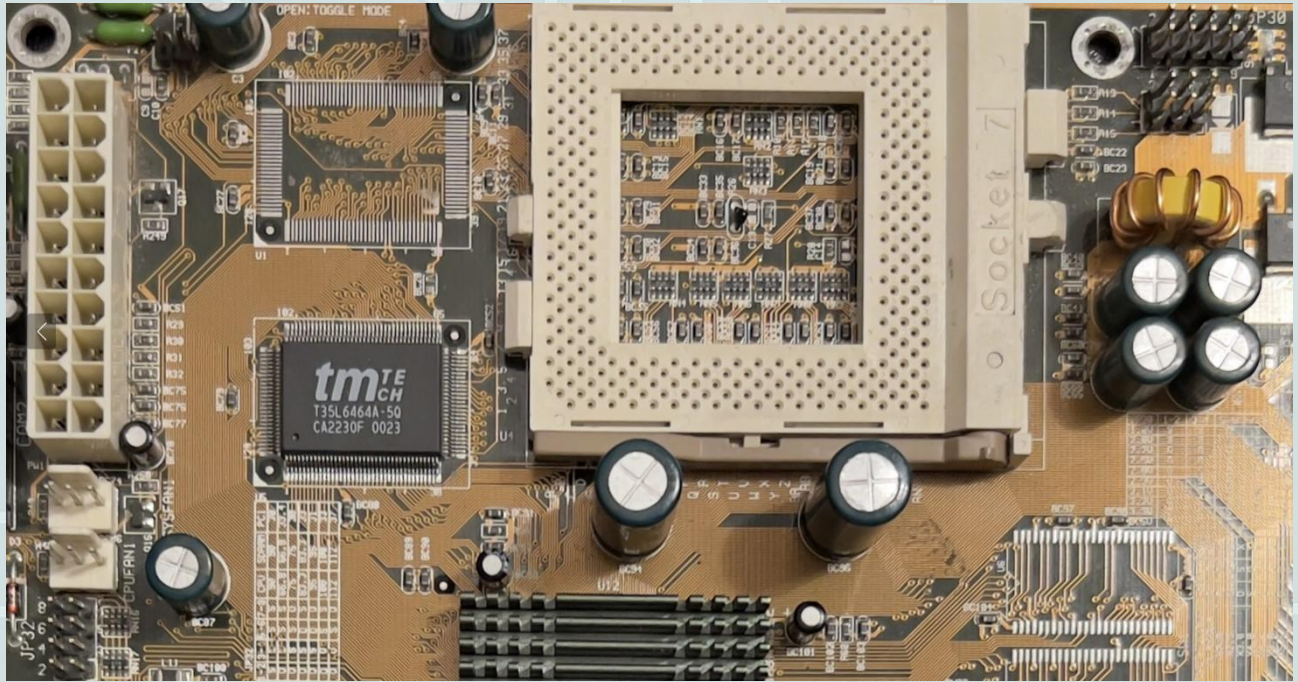
- **Jumper Adjustments:** Many motherboards from the late 80s to early 2000s used jumper blocks to configure system settings such as the Front Side Bus (FSB) speed, CPU multiplier, and voltage. These jumpers—tiny plastic-covered metal bridges—connected specific pairs of pins on the board. By moving the jumpers to different positions according to the motherboard manual, users could select predefined speed settings. For instance, increasing the FSB from 66 MHz to 100 MHz could yield a significant performance gain. In more adventurous cases, users would choose undocumented jumper settings to push the CPU into unsupported frequency ranges, hoping the system would still POST (Power-On Self-Test) successfully. This method required close attention to documentation, trial and error, and a willingness to risk system instability or failure.
- **DIP Switches:** Some motherboards replaced jumpers with DIP (Dual In-line Package) switches—small toggles that could be flipped ON or OFF to configure similar options. DIP switches allowed for slightly easier configuration changes, but they too required detailed reference charts,

typically printed in the motherboard manual or silkscreened onto the PCB. These switches controlled settings such as CPU multiplier ratios and FSB speeds, and experimenting with different switch positions became a rite of passage for early PC hobbyists. Misconfigurations could result in a non-booting system or unpredictable behavior.

- **Clock Generator Modifications:** For more advanced users, modifying the clock generator circuitry itself was an option. The clock generator is responsible for producing the reference frequency for the entire system. By altering the components around this chip—such as soldering additional resistors or capacitors—users could manipulate the output frequency. This kind of hardware hacking was not for the faint of heart, as it often voided warranties and could lead to permanent hardware damage. Nonetheless, for those with the skill, it opened the door to frequencies far beyond what the motherboard officially supported.
- **Voltage Modifications ("VMods"):** Sometimes, achieving stable overclocks required more than just a higher frequency—it demanded additional voltage. When BIOS settings didn't allow for voltage adjustments, overclockers would manually modify the power delivery circuits on the motherboard. These VMods involved soldering resistors or potentiometers directly onto voltage regulation components to raise the Vcore. While this could stabilize higher CPU speeds, it also significantly increased heat output and the risk of damaging the CPU. VMods were considered one of the most dangerous and advanced techniques in the early overclocker's toolkit.

To illustrate how hands-on these processes were, let's take a look at the **Soyo SY-5SSM** motherboard (imagined below) a classic example from the late 1990s. The manual (as can be seen in the screenshot) for this board featured detailed diagrams showing the location and configuration of jumpers responsible for selecting the CPU frequency and multiplier. Setting up a new processor meant consulting these pages, identifying your

CPU model, and carefully configuring the jumpers with a small plastic tool or tweezers. For the adventurous, deviating from the official settings allowed for minor overlocks - enough to squeeze out a few extra megahertz for better performance in DOS games or early Windows applications. This tactile, trial-and-error process was both frustrating and deeply satisfying, and it laid the foundation for the modern overclocking culture we know today.



Hardware Setup**SY-5SSM**

Step 4. CPU Frequency Setting (SW1,JP32)

SW1
Frequency Multiplier

2	4	6
○	○	○
○	○	○
1	3	5

JP32
Host Bus Frequency

8	○	○	7
6	○	○	5
4	○	○	3
2	○	○	1

Modern CPU Features and Their Impact on Clocking

Modern CPUs incorporate several advanced features that can significantly influence overclocking and underclocking:

- **Dynamic Frequency Scaling:** Most contemporary CPUs, such as those with Intel's Turbo Boost technology or AMD's Precision Boost, employ dynamic frequency scaling. This technology allows the CPU to automatically and dynamically adjust its frequency and voltage based on the current workload, thermal conditions, and power limits. When the workload is high, the CPU can boost its frequency to provide increased performance, and when the workload is low, it can reduce its

frequency to save power. Dynamic frequency scaling adds complexity to manual overclocking, as the user may need to disable or fine-tune these automatic mechanisms to achieve a stable and predictable overclock.

- **Voltage Regulators (VRMs):** CPUs receive their power from Voltage Regulator Modules (VRMs) located on the motherboard. VRMs are responsible for converting the motherboard's input voltage to the precise voltage levels required by the CPU. They also play a crucial role in smoothing out voltage fluctuations and providing a stable power supply to the CPU.

The Evolution to BIOS and Software Overclocking

As motherboards and BIOS became more sophisticated, overclocking gradually shifted away from these hardware-based methods. The BIOS started to include more comprehensive options for adjusting CPU frequencies and voltages, making overclocking more accessible and less risky. Software utilities further simplified the process by allowing

users to overclock from within the operating system.

As a real-world example, let's take the BIOS of the fantastic motherboard **EpoX EP-8KRAI** which has the following page dedicated to fine tune the system performance. The support from this motherboard toward overclocking is phenomenal (considering its age).

System Performance	[Standard]
× CPU Timing	Normal
× ROMSIP Table	Normal
× DRAM Command Rate	2T Command
Current FSB Frequency	200 MHz
Current DRAM Frequency	166 MHz
DRAM Clock	[By SPD]
Auto Detect PCI Clk	[Enabled]
Spread Spectrum	[Disabled]
CPU Clock	[200]
CPU Ratio	[Auto]
Next boot AGP/PCI = 66/33 MHz	
Ucore Default Voltage	1.650 V
Current Voltage	[1.700 V]
Adjust Voltage	+ 0.050 V
DIMM Default Voltage	2.70 V
Add Voltage	[+0.20 V]
New Voltage	2.90 V

The first step is to specify in **System Performance** is whether we want the general settings, or one that allows us to select different performance profiles. "Standard" indicates the default settings. Overclocking would typically involve moving away from this standard setting, often by manually configuring the individual options below. Moving from "Standard" to "Expert" (or similar terms like "Performance" or "Manual" depending what you have on your system) typically unlocks more advanced settings and applies a profile that is optimized for performance, often involving changes that facilitate or represent an overclock.

The entries under **CPU Timing**, **ROMSIP Table** and **DRAM Command Rate** are related to memory (RAM) timings. Tighter timings (lower numbers) generally lead to better memory performance. When overclocking, you might adjust these settings to improve memory speed and stability, although overly aggressive timings can cause instability. For example, "Fast" and "Ultra" indicate more aggressive (tighter) timings for CPU and potentially system-level operations compared to "Normal". Tighter timings can improve performance but require more stability and often higher voltages, especially when combined with higher frequencies.

The value specified at **Current FSB Frequency** (for now 200 MHz) is the Front Side Bus Frequency, which is a key component of the overall CPU speed, particularly on older architectures. The CPU speed is often calculated as FSB Frequency multiplied by the CPU Ratio. Increasing the FSB is a common way to overclock the CPU and also affects the speed of other components like the RAM. Changing it for example to 100, would be a major difference because the FSB frequency would significantly be lower in this configuration. On older systems, increasing the FSB was a primary method of overclocking the CPU and memory. A lower FSB here

suggests the system is relying more on the CPU multiplier to achieve speed.

The **Current DRAM Frequency** (for now 166 MHz) shows the current operating speed of your RAM. The DRAM frequency is often linked to the FSB frequency through a multiplier or divider. Increasing the DRAM frequency can improve system performance, especially in memory-intensive tasks.

The default setting from **DRAM Clock** is By SPD (Serial Presence Detect), which is information stored on the RAM modules that defines their standard operating speeds and timings. "By SPD" means the system is automatically configuring the RAM based on this information. For overclocking, you often need to manually set the DRAM frequency and timings instead of relying on SPD to push the RAM beyond its standard specifications. If for example we could modify the DRAM frequency to a lower configuration, which should be consistent with the lower FSB (as memory speed is often linked to FSB). For example, if the DRAM clock is manually set to 133 MHz instead of "By SPD", indicating manual control over memory speed, even though it's a lower frequency than would be provided by default, which might be paired with very tight manual timings to compensate for the lower frequency, it would simply reflect a system where memory speed isn't being pushed as hard as the CPU, and this might lead to a lower performance system, than the default settings.

The **Auto Detect PCI Clk** relates to the PCI bus clock speed. It's usually best left enabled unless you encounter specific compatibility issues.

Interesting is the **Spread Spectrum** setting, which is a technique used to reduce electromagnetic interference (EMI) by slightly varying the clock frequencies. When overclocking, it's typically disabled because it can sometimes introduce minor instability at

high frequencies. Disabling it provides a more stable clock signal. Enabling Spread Spectrum slightly varies the clock signal to reduce EMI. While harmless in standard operation, it can sometimes introduce minor instability at very high frequencies when overclocking. It's generally recommended to disable it for maximum overclocking stability. Having it enabled in an "Expert" performance profile could be somewhat counter-intuitive from a stability-focused overclocking perspective.

The **CPU Clock** field shows the calculated CPU speed. It appears to be indicating that the CPU clock is derived from the FSB (200 MHz) multiplied by an automatic ratio.

The **CPU Ratio** should already be familiar to us: this is the multiplier that, when combined with the FSB, determines the final CPU clock speed. If the CPU has an unlocked multiplier, you can manually increase this ratio to overclock the CPU without changing the FSB. "[Auto]" means the system is automatically setting the ratio, likely to the CPU's default or maximum turbo speed. Overclocking a CPU with an unlocked multiplier involves increasing this value. Setting the CPU ratio for example to 20, and the CPU clock to 100 would clearly show the CPU speed is calculated as 100 MHz (FSB) multiplied by a ratio of 20, resulting in a 2000 MHz (2 GHz) CPU speed, thus achieving the same 200 MHz FSB as with an "Auto" ratio.

The next part of the BIOS is at a much more technical level, and it involves in the scary sounding step of modifying the voltage of the system.

Vcore Default Voltage [1.650 V], **Current Voltage** [1.700 V] and **Adjust Voltage** [+0.050 V] all play their part in providing the CPU the much-needed electricity. Vcore is the core voltage supplied to the CPU. Keep in mind, higher clock speeds often require more voltage to remain stable. The image shows the default voltage, the current voltage, and an option to adjust it (in this case, an additional 0.050V is being added, resulting in a current voltage of 1.700V). Increasing Vcore is a critical part of overclocking for stability, but it also significantly increases heat output and can potentially damage the CPU if set too high.

The next section is related to providing electricity to the RAM modules through the settings **DIMM Default Voltage** [2.70 V], **Add Voltage** [+0.20 V] and **New Voltage** [2.90 V]. DIMM voltage is the voltage supplied to the RAM modules. Similar to the CPU, increasing the RAM voltage can improve stability when running the memory at higher frequencies or tighter timings than their standard specifications. The image shows the default voltage, an added voltage of 0.20V, resulting in a new voltage of 2.90V. Increasing DIMM voltage can also increase heat and potentially damage the RAM if set too high.

The Electrical Theory of CPU Clocking

To truly understand the effects of overclocking and underclocking, it's essential to delve into the underlying electrical principles that govern CPU operation.

Modern CPUs are built using Complementary Metal-Oxide-Semiconductor (CMOS) logic gates. These gates are the fundamental building blocks of digital circuits, and they control the flow of electrical current to perform

logical operations. CMOS gates use pairs of transistors, specifically NMOS (N-type Metal-Oxide-Semiconductor) and PMOS (P-type Metal-Oxide-Semiconductor) transistors, to switch between ON and OFF states, representing binary 1s and 0s. The speed at which these gates can switch between states directly impacts the CPU's clock speed and its ability to process information. Several factors govern this switching speed:

Every transistor gate possesses a small amount of capacitance, which is the ability to store electrical charge. This capacitance must be charged or discharged for the transistor to switch states. The higher the capacitance, the more time it takes to charge or discharge, and the slower the switching speed.

The voltage applied to the transistor gate influences the speed at which the gate capacitance can be charged or discharged. A higher voltage allows for faster charging and discharging, leading to quicker switching. This is why increasing the CPU core voltage (V_{core}) can enable higher clock speeds in overclocking.

The amount of electrical current available to charge or discharge the gate capacitance also affects switching speed. A larger current allows for faster charging and discharging, contributing to quicker switching. The switching time (τ) of a CMOS gate is approximately proportional to the product of the resistance (R) and capacitance (C) in the circuit ($\tau \approx RC$). This equation highlights that to achieve faster switching and thus higher clock speeds, either the capacitance must be reduced (which is generally fixed in the physical design of the silicon chip) or the voltage (V_{core}) must be increased to supply more current.

The power consumed by a CPU is a critical factor in its operation and is directly related to both its clock frequency and operating voltage. The power consumption (P) of a CPU can be approximated by the following equation:

$$P = C \times V^2 \times f$$

where:

- P is the power consumed, measured in watts.
- C is the total switched capacitance of the CPU's transistors.
- V is the supply voltage (V_{core}).
- f is the clock frequency.

This equation reveals two crucial relationships:

As the clock frequency (f) increases, the power consumption (P) also increases proportionally. This means that doubling the clock speed roughly doubles the power consumption. The power consumption (P) increases much more rapidly with increases in the supply voltage (V). Doubling the voltage quadruples the power consumption. This quadratic relationship explains why increasing the V_{core} to stabilize higher clock frequencies in overclocking leads to a substantial increase in heat generation.

Electrical power that is not converted into useful work by the CPU is dissipated as heat. CPUs have specific operating temperature ranges, and exceeding these limits can have detrimental consequences. Typically, CPUs are designed to operate safely below temperatures of around 90-100°C. Excessive heat can lead to:

To protect itself from damage, the CPU may automatically reduce its clock speed and voltage when it detects that its temperature is exceeding safe limits. This process, known as thermal throttling, results in a decrease in performance.

Overheating can cause the CPU to become unstable, leading to system crashes, data corruption, and unpredictable behavior.

Prolonged exposure to high temperatures can cause permanent damage to the CPU's silicon components, reducing its lifespan or rendering it unusable. To mitigate the risks associated with heat generation, effective cooling solutions are essential, especially in overclocked systems. These solutions include:

- **Fans:** Fans are used to circulate air and dissipate heat away from the CPU heatsink.
- **Heat Pipes:** Heat pipes are highly efficient thermal conductors that transfer heat away from the CPU to a heatsink where it can be dissipated by airflow.

- **Liquid Cooling:** Liquid cooling systems use a liquid coolant to absorb heat from the CPU and transfer it to a radiator, where it is dissipated by fans. Liquid cooling provides superior cooling performance compared to air cooling but is more complex and expensive.

Risks and Benefits

Both overclocking and underclocking offer potential benefits but also carry certain risks that users should carefully consider:

Overclocking Risks

- **Instability:** One of the most common risks of overclocking is system instability. This can manifest as frequent crashes, blue screens of death, data corruption, and unpredictable system behavior. Overclocking pushes the CPU beyond its designed operating parameters, which can lead to errors and instability if not done correctly.
- **Thermal Stress:** Overclocking increases heat generation, which can put significant thermal stress on the CPU and other system components. Excessive heat can accelerate the degradation of components over time, potentially shortening their lifespan.
- **Higher Power Draw:** Overclocking requires the CPU to draw more power, which can strain the motherboard's power

delivery system and the power supply unit. This increased power draw can potentially damage these components if they are not designed to handle it.

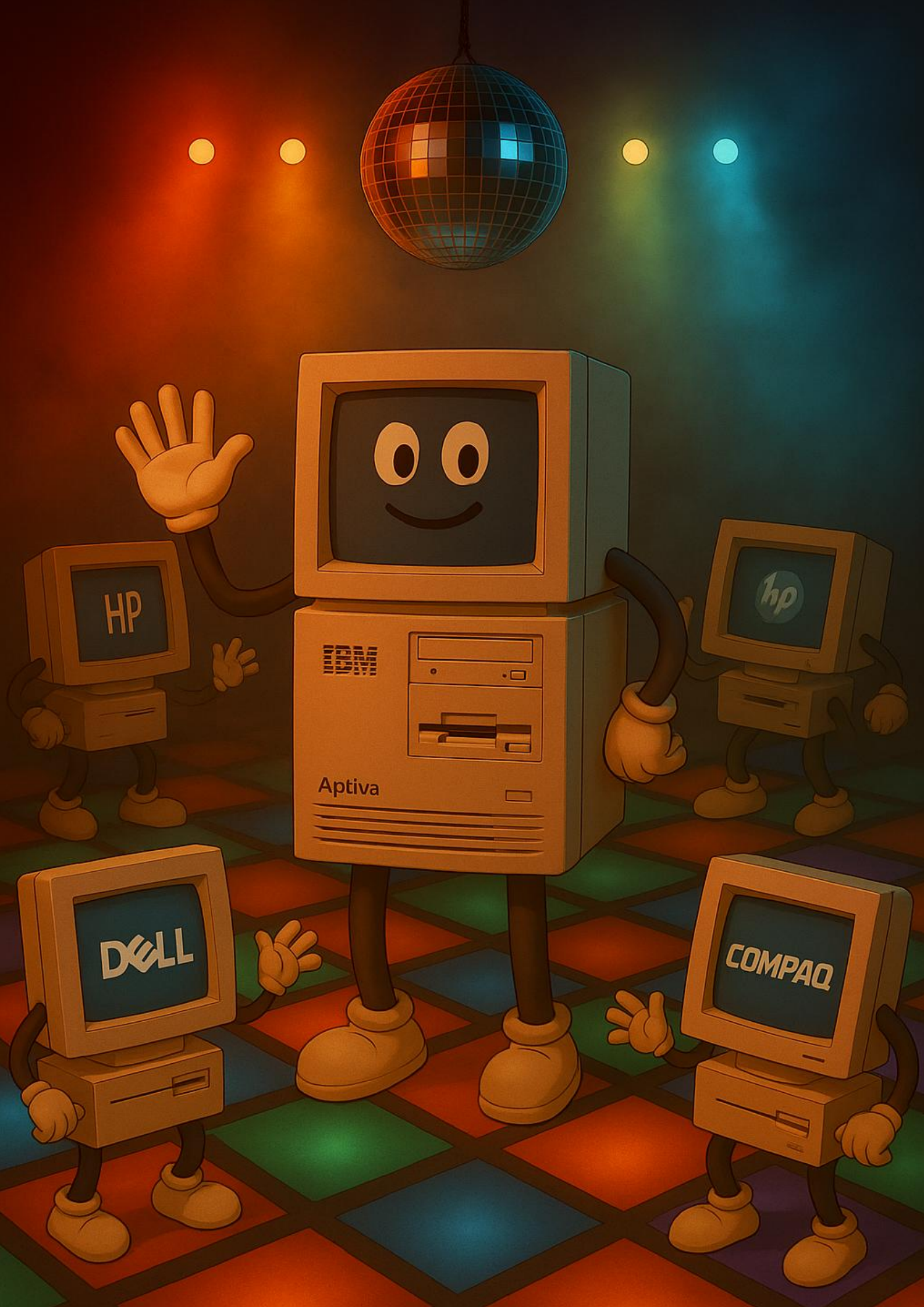
- **Voided Warranty:** Overclocking typically voids the manufacturer's warranty on the CPU and potentially other components. If damage occurs due to overclocking, the user will be responsible for the cost of repairs or replacements.

Underclocking Risks

- **Reduced Performance:** The most obvious risk of underclocking is a reduction in overall system performance. While underclocking can improve energy efficiency and reduce heat, it comes at the cost of lower processing speeds, which can make the system feel slower and less responsive, especially in demanding applications.
- **Incompatibilities:** In some cases, underclocking can lead to incompatibilities with certain hardware or software that are designed to operate at the CPU's default speeds. Some applications may rely on specific timing or performance characteristics of the CPU, and underclocking can disrupt these expectations, leading to errors or malfunctions.

Closing words

As we've seen, the path to overclocking or underclocking is paved with careful adjustments, informed choices, and a solid understanding of both hardware capabilities and limitations. From early jumper tweaks to advanced BIOS configurations, the principles remain the same: balance performance, stability, and thermal constraints to suit your needs. Whether you're reviving retro systems or optimizing modern setups, the tools and knowledge are now more accessible than ever. With thoughtful tuning and a respect for the risks involved, you're well-equipped to shape your system's behavior - one clock cycle at a time.



One Aptiva to rule them all,
One Aptiva to find them,
One Aptiva to bring them all
and in the circuits bind them,



In the Land of Living Room
where the dial-tones call.

THE IBM APTIVA

THE LAST DANCE OF IBM IN THE HOME PC DISCO

Iack in the shimmering twilight of the 1990s, when AOL ruled the Internet and your computer made noises like a squirrel in pain to just to get online, **IBM** decided it was going to take one more heroic swing at the consumer desktop market. Its weapon of choice? The Series of IBM Aptiva machines, gallant, well-groomed knights in (mostly) beige armor riding an Intel-powered steed ... except those that used AMD.

Our machine comes from an abandoned barn, left there by the previous owners and is a typical representative of that specific era. It is an IBM Aptiva 2139-E5Y (a surprising serial number, I must admit, since it is not in the official serial numbers of the machine, maybe it was a strange Nordic concoction), but before we unpack the Aptiva 2139's quirks and charms, let's rewind a bit.

A LITTLE APTIVA HISTORY

IBM, the godfather of the personal computer, had always been that stern uncle at the family BBQ - powerful, respected, but not exactly fun. The original IBM PC, released in 1981, was a beige brick of serious business, and it dominated corporate America. But by the early '90s, that party had moved on. Kids were playing games. Families were emailing each other. And Bill Gates was dropping Windows like it was hot.



His Majesty, the IBM Aptiva 2139

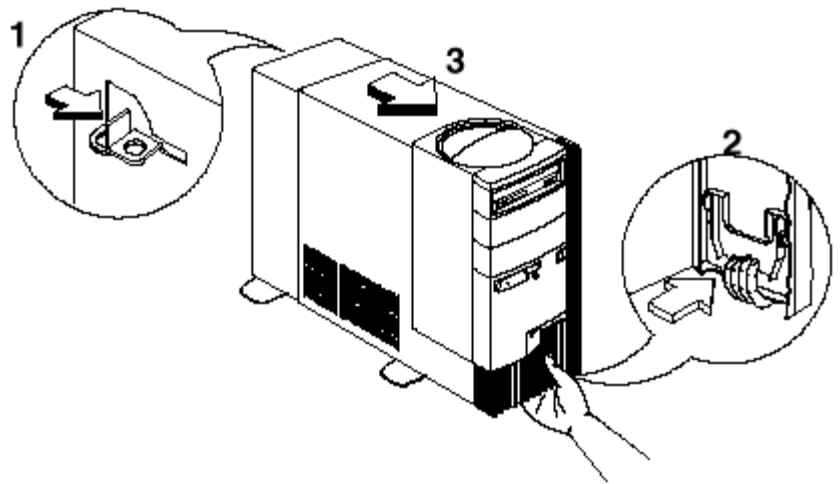
IBM had already dipped its toes in the home market with the PS/1 (which seemingly nobody remembers, maybe on purpose), but in 1994, they brought out the Aptiva line. This was IBM's attempt to say, "Hey, we can be fun too!" And for a while, it kinda worked. Aptiva machines came loaded with multimedia software, CD-ROM drives (which were magical back then), and even kid-friendly programs. Some early models shipped with IBM's own OS/2 Warp, a noble but doomed operating system that most people replaced with Windows faster than you could say "start menu".

Fast forward to the year 1999. The Y2K bug was looming, the internet was slow, like a rheumatic snail, and everyone had a friend who swore their computer got hacked because they clicked a dancing baby GIF. IBM, now in its late-stage consumer PC era, released the Aptiva 2139 series.

The 2139 wasn't exactly a Ferrari, but it was a reliable four-door sedan with a good stereo and a cupholder. It ran either on an AMD K6-2 processor at 450 MHz, or as the one we present here and now an Intel Pentium II. Both of these were paired with an ATI Rage Pro, which surprisingly was fast enough to run Quake II, play a CD, and print a book report simultaneously - if you were lucky. It usually came with 64 megabytes of RAM, which, at the time, felt like unlimited power, however we managed to upgrade ours to 512 because that is the choking point of Windows 98, just out of the box. Kids today might scoff at that, but remember: back then, after being seasoned on DOS we were amazed that a computer could even show a photo and play a sound at the same time.



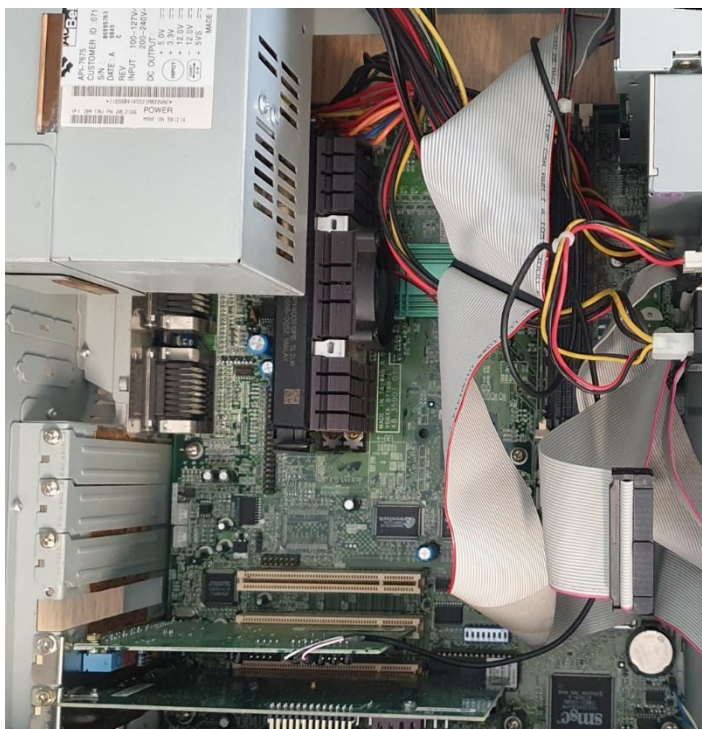
This machine typically had a 10 GB hard drive - enough to store every MP3 you could download on Napster before your parents picked up the phone and killed the dial-up connection, ours is upgraded to 20GB. Since I was not extraordinarily impressed by the way graphics were handled by the ATI Rage Pro, I have stuffed in mine a Voodoo3, the PCI edition, since our machine is the “Entry” level version, whose motherboard is provided by Acer, hence no AGP slot on it. Sadly. Considering that a decent period correct PCI Riva TNT costs and arm and a leg, the Voodoo 3 with its 16 MB was the best possible choice for the PCI port and this machine.



Removing the cover can be tricky, even if we follow the manual's steps

The 2139 had everything a late-'90s family could want: a CD-ROM drive, a 3.5-inch floppy disk (because some people still trusted those), you could stuff in even a 56k modem that connected you to the Information Superhighway at roughly the speed of molasses. USB ports were there too - well, a couple of them - and they worked... eventually after installing the proper driver.

The IBM Aptiva 2139 has a front plate so unmistakable, it practically screams, “I’m from the '90s, and I *mean* business”. Its design is a glorious symphony of curves and plastic - bold,



beige, and utterly unforgettable.

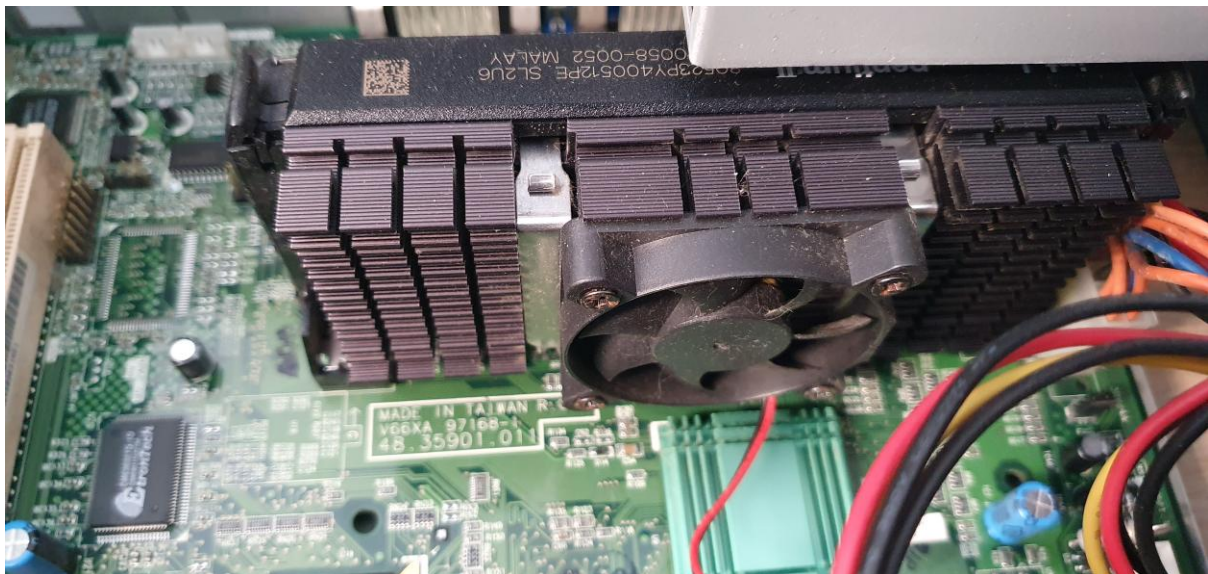
Unfortunately, nestled among its majestic facade were two gaping 5.25" drive bays, the machine was delivered to me with. Now, you’d think filling these would be easy. Nope. IBM, in its infinite wisdom, used **drive bay covers so proprietary**, they might as well have been forged in the fires of Mount Doom. To this day, I’ve only ever seen two in the wild, and one of them was in a blurry eBay listing that turned out to be a microwave. The other elusive bay cover? I did spot it tucked snugly into another Aptiva, proudly displayed on a

Top View of the Motherboard – Or better said, Top View of the ribbon cable chaos

vintage computing forum like it was the crown jewel of the Smithsonian. I messaged the owner, hopeful, desperate, offering trades, bribes, perhaps even half of my soul. His reply was swift and cold:

"Over my dead body. Or a 3DFx Voodoo 5 card..."

So, what did I do? I improvised. I sacrificed aesthetics for pragmatism and **stuffed in two extra CD drives**, purely to plug the void. Do they work? Absolutely not. Do they light up? Never. They are not even plugged in. But from the front, at least, it looks like my Aptiva means serious multi-media business. It's all for the illusion — because nothing says "I have my life together" like a computer with three CD-ROM drives and zero shame.



The 400Mhz Pentium II in all its glory, sprinkled with a little dust

SOFTWARE AND STYLE

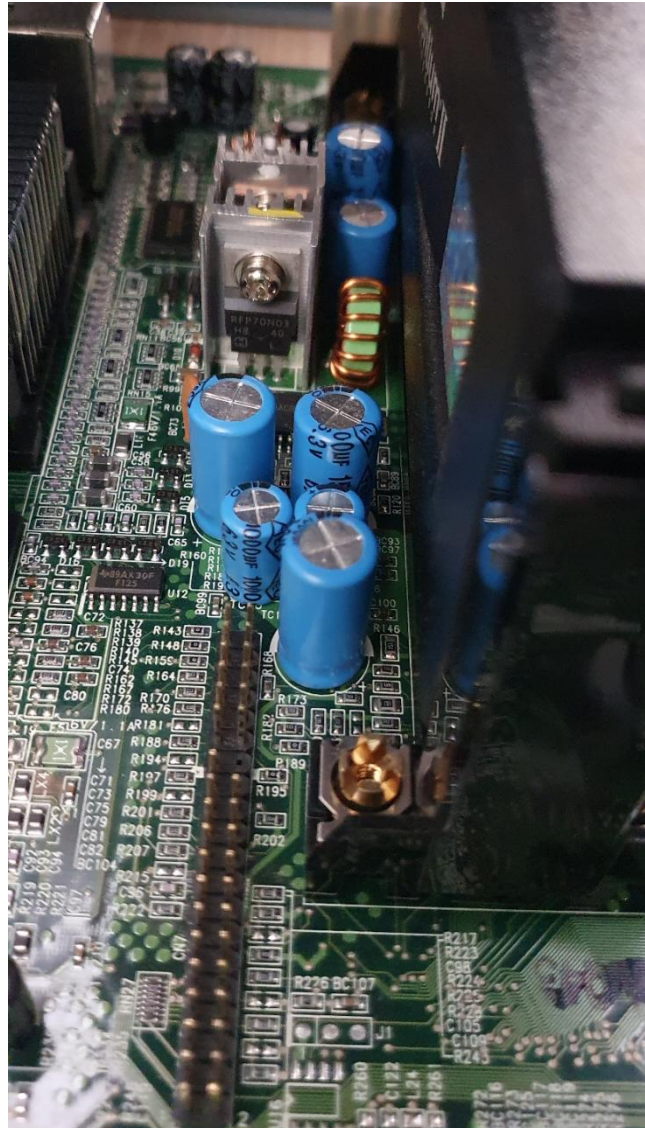
What really made the Aptiva line unique wasn't just the hardware. It was IBM's approach at software and accessories. IBM cemented its Model M keyboard as one of the most sought-after ones, at least in recent years and while these machines came with IBM's "Rapid Access" keyboard, complete with shortcut buttons labeled "Internet," "Help," and "Email" — the offering was nothing of a short.

IBM included bundled programs like Lotus SmartSuite (IBM's own Microsoft Office-alternative that offered a timely alternative to Microsoft's) and games that were probably educational, though most people just installed SimCity 2000 and called it a day. Seemingly the machine was very popular in Norway, since the only rescue and restore CDs the almighty archive holds at the moment of writing are the ones that are in the Norwegian language.

SO WHAT HAPPENED

Despite IBM's efforts, the Aptiva line couldn't keep up with the wild price cuts and flashy marketing of competitors like Compaq, Gateway, and HP. Dell, in particular, was revolutionizing the direct-to-consumer model, letting buyers customize PCs online and have them shipped straight to their doors. IBM was still selling through traditional retail, often at higher prices, and frankly, regardless the high quality (because what else would you call that the machine from 1998 still works in 2025), most people went for the cheap and fancy boxes they “upgraded” from in a few months regardless.

By 2000, IBM gracefully bowed out of the home PC business in the U.S. The Aptiva brand was quietly retired, and IBM turned its attention back to enterprise hardware and ThinkPads - leaving behind a dusty beige legacy in dens and attics across the worlds.





C

**PROGRAMMING
TUTORIAL
PART 2**

Introduction

In the C programming language, managing and organizing data efficiently is crucial for building robust and scalable applications. While basic data types such as `int`, `float`, and `char` are suitable for simple values, real-world problems often require grouping multiple values of different types into a single entity. This is where structures come into play.

Structures (`struct`) provide a powerful and flexible way to create user-defined data types that can encapsulate multiple attributes, making your code more readable and logically organized. Whether you're building a simple contact list or working with complex records in memory, understanding structures is a fundamental skill for any C programmer.

This current iteration of the C tutorial introduces the concept of structures in C, explaining their declaration, initialization, and usage. It also covers advanced topics such as dynamic memory allocation, pointers to structures, and arrays of structures, complete with practical examples and annotated code snippets to help reinforce the concepts.

From the technical side, feel free to use the VSCode setup we created last time, or if you wish to use a different compiler that is also perfectly fine. So, let's start coding.

Structures in C

Structures are user-defined data types that allow you to group together variables of different data types under a single name. This is useful when you want to represent something more complex than a single number or character — for example, a person, a product, or a point in space.

Declaring Structures

To declare a structure, use the `struct` keyword followed by a name and a block containing member declarations.

The example on the right side defines a `Person` structure that contains a name (as a string), an age (as an integer), and a height (as a floating-point number). We use `char name[50]` instead of a pointer to a string because it allows us to allocate space directly within the structure, making it simpler to manage for beginners.

Why group these together? Because these attributes all belong to a single logical entity — a person. Using a structure allows you to manage and pass around this collection of attributes as a single variable.

```
1. struct Person
2. {
3.     char name[50];
4.     int age;
5.     float height;
6. };
```

Creating and Initializing Structure Variables

Once declared, you can create variables of the structure type:

```
1. struct Person p1; // Creates a variable p1 of type Person
```

You can also initialize a structure during declaration:

```
1. struct Person p1 = {"Alice", 25, 5.7};
```

This is called the positional initialization, and using this, we set the initial values of the structure's members in the order they were declared in the structure declaration.

There are **several other ways** to initialize a `struct Person` variable, and each method we present has its advantages depending on what you need.

Designated Initializers (C99 and later)

You can initialize specific fields by name, and **the order doesn't matter**:

```
1. struct Person alice = { .name = "Alice", .age = 30, .height = 1.65f };
```

- **Advantages:** Clear, flexible, and easy to read.
- You can even skip fields if needed; skipped fields are **zeroed** automatically.

A Partial initialization looks like:

```
1. struct Person bob = { .name = "Bob", .height = 1.80f };
```

Here, age will be initialized to 0.

Partial Initialization by Position

If you provide **only some** values in positional initialization, the missing fields are automatically **set to zero**:

```
1. struct Person diana = { "Diana" };
```

- Only the name is initialized.
- age will be 0, and height will be 0.0f.

Zero Initialization

Sometimes you want to start with a structure **entirely filled with zeros** (empty strings, zero integers, zero floats):

```
1. struct Person empty_person = { 0 };
```

or (in C99):

```
1. struct Person empty_person = { };
```

- Every field is initialized to **zero**.
- Useful when you want a **clean slate**.

Compound Literals (C99 and later)

Compound literals let you **create and initialize** a structure **on the fly**:

```
1. struct Person frank = (struct Person){ "Frank", 45, 1.75f };
```

or using designated fields:

```
1. struct Person grace = (struct Person){.name = "Grace", .age = 35, .height = 1.68f };
```

- **Useful** when you need to pass an initialized structure directly into a function or for quick one-off uses.

Accessing Structure Members

To access individual members, use the dot operator:

```
1. strcpy(p1.name, "Alice");
2. p1.age = 25;
3. p1.height = 5.7;
```

We use `strcpy` to assign a string because arrays in C cannot be assigned using the `=` operator.

Example Program

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. struct Person {
5.     char name[50];
6.     int age;
7.     float height;
8. };
9.
10. int main() {
11.     struct Person p1 = {"Alice", 25, 5.7};
12.     printf("Name: %s, Age: %d, Height: %.1f\n",
13.           p1.name, p1.age, p1.height);
14.     return 0;
15. }
```

This simple program demonstrates how to define a structure, initialize it, and access its members.

With the skills you've gained so far, you're already capable of creating programs that can make a real impact.

But this is just the beginning!

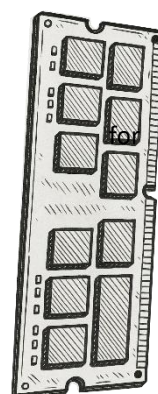
By learning a few more key concepts - like memory management and other powerful techniques - you'll unlock even greater possibilities.

Keep going; you're building a strong foundation for something amazing!

Dynamic Memory Allocation

Before we talk about how to use dynamic memory allocation in C, let's understand what memory means in a program. When you run a C program, your computer sets aside a certain amount of memory (RAM) it. This memory is divided into different sections:

- **Stack:** This is where local variables (declared inside functions) are stored. The size of this memory is usually fixed and limited.
- **Heap:** This is a large area of memory used for dynamic allocation - memory that you can request while the program is running.



Dynamic memory allocation is useful when you don't know ahead of time how much memory your program will need. For example, if you're reading a list of names from a file and you don't know how many names there will be, you can't safely declare a fixed-size array. Instead, you can request just the amount of memory you need - and adjust it as the program runs.

C provides several functions to manage memory on the heap:

- `malloc()` – Allocates a block of memory.
- `calloc()` – Allocates and clears a block of memory.
- `realloc()` – Resizes a previously allocated block.
- `free()` – Releases memory when you're done using it.

Using malloc

The `malloc()` function (short for memory allocation) reserves a block of memory of the size you specify. It returns a pointer to the beginning of that memory block.

```
1. #include <stdlib.h>
2. int *ptr = (int*)malloc(5 * sizeof(int));
```

In this example, `malloc` allocates enough memory for five integers. `sizeof(int)` ensures the memory size matches your system's requirements. We cast the result to an `int*` so we can use it like a pointer to integers.

After allocation, always check if it succeeded:

```
1. if (ptr == NULL) {
2.     printf("Memory allocation failed\n");
3.     return 1;
4. }
```

Once you've allocated memory, you can use it just like an array:

```
1. for (int i = 0; i < 5; i++) {
2.     ptr[i] = i * 10;
3. }
4.
5. for (int i = 0; i < 5; i++) {
6.     printf("%d ", ptr[i]);
7. }
```

This creates an array in memory that holds values like 0, 10, 20, 30, 40.

Freeing Memory

When you're done using dynamically allocated memory, you must release it back to the system using the `free()` function:

```
1. free(ptr);
```

This prevents memory leaks, which happen when a program uses memory but never gives it back. Over time, especially in long-running applications, memory leaks can cause your system to slow down or crash.

Changing the Size with realloc

In certain situations, it might happen that the memory allocated is not the right one. It's either too much, or too less. If you need more (or less) space later, use `realloc()`:

```
1. ptr = (int*)realloc(ptr, 10 * sizeof(int));
```

This changes the size of the memory block to hold ten integers instead of five. `realloc` tries to keep existing data if possible and move it to a new location if needed.

Again, always check that it succeeded:

```
1. if (ptr == NULL) {
2.     printf("Memory reallocation failed\n");
3.     return 1;
4. }
```

Dynamic memory gives your programs flexibility and power - you can create data structures that grow and shrink as needed, based on the input or user behavior. This is essential for working with real-world data, where nothing is truly fixed. However, understanding this topic is also one of the more complex one when it comes about programming. So please dwell on it as long as you feel you have a true mastery of the topic.

Pointers to Structures

So far, we've accessed structure members using the dot (.) operator, like `alice.age`. But in many real-world programs, especially those involving dynamic memory or function calls, we often deal with **pointers to structures** instead.

A pointer is a variable that stores the memory address of another variable. When it comes to structures, pointers allow you to:

- Work with dynamically allocated structures.
- Pass large structures efficiently to functions (without making copies).
- Modify the original structure from within a function.

Understanding pointers to structures is quite important for building more flexible and memory-efficient programs in C.

Declaring and Assigning Pointers

You can create a pointer to a structure just like any other pointer:

```
1. struct Person p1 = {"Alice", 25, 5.7};
2. struct Person *ptr = &p1;
```

In this example, `ptr` holds the memory address of the structure variable `p1`. The `&` operator is used to get the address of `p1`.

Accessing Structure Members via Pointers

To access members through a structure pointer, use the arrow operator (`->`).

```
1. printf("Name: %s\n", ptr->name);
2. ptr->age = 26;
```

This is equivalent to:

```
1. (*ptr).age = 26;
```

But the arrow syntax is shorter and more readable. The parentheses are necessary in the second form to ensure the member is accessed from the dereferenced structure pointer.

What exactly is dereferencing in this case? Simply said, let's imagine this step by step - no jargon for now.

In C, a **pointer** is a variable that **stores a memory address**. It's like saying, "Hey, the thing you care about is *over there* at this location." Now, **dereferencing a pointer** simply means: **Go to that address and get (or change) the actual value that's stored there**. Or, using an analogy: think of a pointer like a house address written on a piece of paper. You can't live inside the piece of paper - you have to **use the address to go to the house**. Similarly, you can't do much with the pointer itself if you want the real data - you **dereference** the pointer to **access the real thing** in memory.

In short: Dereferencing a pointer means following the address to get the actual value stored there.

Dynamically Allocating Structures

Just like arrays or primitive types, structures can also be dynamically allocated using `malloc` or `calloc`. This is especially useful when the structure needs to live beyond the scope of a function, or when the number of structures isn't known in advance.

```
1. struct Person *ptr = (struct Person*)malloc(sizeof(struct Person));
2. if (ptr == NULL) {
3.     printf("Memory allocation failed\n");
4.     return 1;
5. }
```

This code dynamically allocates memory for one `Person` structure and stores its address in the pointer `ptr`. The memory comes from the heap, so it remains available until explicitly freed.

You can now use the pointer to set values:

```
1. strcpy(ptr->name, "Diana");
2. ptr->age = 22;
3. ptr->height = 5.5;
```

And remember to free the memory when you're done with:

```
1. free(ptr);
```

in order to release back to the care of the operating system. From this point on `ptr` is unusable for your application, and if you try to use it you will possibly run into one of those famous "Run After Use" scenarios which highly possibly will result with a crash of your application, which unless your program is SimCity and runs under Windows95 is never a good situation for your program to be in.

Why Use Dynamically Allocated Structures?

Using malloc with structures gives you more control. For example, you can:

- Create structures at runtime based on user input.
- Store structures in dynamically allocated arrays or linked lists.
- Return structures from functions without worrying about scope.

Example Program: Dynamic Structure Allocation

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4.
5. struct Person {
6.     char name[50];
7.     int age;
8.     float height;
9. };
10.
11. int main() {
12.     struct Person *ptr = (struct Person*)malloc(sizeof(struct Person));
13.
14.     if (ptr == NULL) {
15.         printf("Memory allocation failed");
16.         return 1;
17.     }
18.
19.     strcpy(ptr->name, "Diana");
20.     ptr->age = 22;
21.     ptr->height = 5.5;
22.
23.     printf("Name: %s\n", ptr->name);
24.     printf("Age: %d\n", ptr->age);
25.     printf("Height: %.1f\n", ptr->height);
26.
27.     free(ptr); // Always free dynamically allocated memory
28.     return 0;
29. }
```

This program demonstrates how to allocate, initialize, and use a structure in dynamic memory. It's a common pattern you'll use when working with larger programs that manage data at runtime.

Pointers to structures give you the tools you need to write memory-efficient, modular C code. Once you're comfortable with them, they open the door to advanced features like linked lists, trees, dynamic arrays, and even basic object-oriented techniques in C.

Arrays of Structures

Sometimes you need to store a group of similar structures — like a list of people, a set of books, or multiple sensor readings. Arrays of structures make this easy. You can think of them as a list where each item is a full structure.

Declaration and Initialization

Here's how to declare an array of structures and initialize it:

```
1. struct Person people[3] = {
2.     {"Alice", 25, 5.7},
3.     {"Bob", 30, 6.0},
4.     {"Charlie", 28, 5.9}
5. }
```

Each element in the array is a separate Person structure. This is helpful when you know in advance how many items you'll need. You can access and modify each element just like you would in a regular array.

Accessing and Modifying Elements

You use the array index followed by the dot operator to access individual members:

```
1. printf("First person's name: %s\n", people[0].name);
2. people[1].age = 31;
```

This is similar to working with basic arrays, but each element holds multiple related values.

Iterating Over the Array

A loop can be used to display or process all the elements:

```
1. for (int i = 0; i < 3; i++) {
2.     printf("%s is %d years old and %.1f feet tall",
3.         people[i].name, people[i].age, people[i].height);
4. }
```

This pattern is extremely useful when processing groups of items.

Using Pointers to Access Array Elements

You can also use a pointer to the first element of the array. In C, the name of the array is a pointer to its first element, so:

```
1. struct Person *ptr = people;
```

Now `ptr` points to the same memory as `people[0]`. You can use pointer arithmetic and the arrow operator to move through and access elements:

```
1. printf("Second person's age: %d\n", (ptr + 1)->age);
```

This is particularly useful in functions or when working with dynamically allocated arrays.

Dynamically Allocated Arrays of Structures

What if you don't know how many items you'll need until the program is running? You can allocate memory for an array of structures dynamically using `malloc`:

```
1. int count = 5;
2. struct Person *people = (struct Person*)malloc(count * sizeof(struct Person));
3. if (people == NULL) {
4.     printf("Memory allocation failed\n");
5.     return 1;
6. }
```

You now have space for `count` number of `Person` structures. You can fill and access them just like a regular array:

```
7. strcpy(people[0].name, "Eve");
8. people[0].age = 24;
9. people[0].height = 5.6;
```

When done, don't forget to free the memory:

```
10. free(people);
```

Why You Can Use C Arrays Like Pointers (and Vice Versa)

You might have observed, that C has a decent flexibility when it comes about handling arrays and pointers. Indeed, this is the case: In C, arrays and pointers are closely related — but they are not the same thing. They behave similarly in many cases because of how C treats arrays when you use them in expressions.

Here's the core rule:

In most expressions, the name of an array automatically "decays" (turns) into a pointer to its first element.

While this might sound daunting in the first, after a quick explanation it will be much more easier, so let's break it down for all to understand.

When you create an array (for example, of three integers):

- The array is a block of memory containing all the elements.
- But when you use the array's name in most expressions, C treats it as a pointer to the first element.

So:

Expression	Meaning
<code>Array_name</code>	Address of the first element
<code>*Array_name</code>	Value of the first element
<code>Array_name + 1</code>	Address of the second element
<code>*(Array_name + 1)</code>	Value of the second element

Why Pointers and Arrays Behave Similarly

C was designed to be very close to the machine — and to keep things fast and simple:

- An array in memory is just a block of consecutive memory cells.
- A pointer is just an address — it points to some place in memory.

So, using a pointer to move through the array is exactly how the array naturally lives in memory — element after element, one after the other.

Thus, it makes sense to let the array name behave like a pointer to the start of the array.

But Arrays and Pointers Are Not the Same!

They behave similarly when used in expressions.

They are different in some important situations:

Aspect	Arrays	Pointers
Memory	Space for all elements is reserved	Only space for the pointer itself
Reassignment	Cannot change an array to point elsewhere	Pointer can be reassigned to point somewhere else
Size	<code>sizeof(array)</code> = total size (e.g., $3 \times 4 = 12$ bytes)	<code>sizeof(pointer)</code> = size of a pointer (typically 4 or 8 bytes)

Quick Visual

Imagine memory like this:

- The first element of the array is at one address.
- The second element immediately follows.
- The third element follows that.

When you use the array name, you're pointing to the first element, and moving through the array is like stepping from one address to the next.

A Friendly Analogy

- Imagine an array is like a train with 3 cars: 🚂🚃🚃
- The array name is a pointer to the first car.
- If you move (+1), you go to the next car.
- A pointer is like having a ticket pointing to some car; you can also move it around.
- But the train itself (array) stays fixed — its cars are built in order and you can't just "move" the train somewhere else!

And putting all this in one place, using even more simpler terms. As a summary: In C, arrays and pointers feel very similar because the name of an array automatically acts like a pointer to its first element. This means you can use pointer arithmetic to move through an array, treating the array name like an address in memory. However, arrays and pointers are not exactly the same: an array is a fixed block of memory, while a pointer is a variable that holds an address and can be changed to point elsewhere. This design makes C arrays and pointers close to how computers really handle memory, but it's important to remember that arrays themselves can't be reassigned like pointers can.

Exercise: Build Your Own Contact Book

Here's a quick and fun mini-project to reinforce what you've learned. You'll build a simple contact book that stores and displays a list of people using structures, arrays, and pointers.

Your Challenge:

1. **Define a structure** called `Contact` with the following members:
 - o `char name[50]`
 - o `char phone[20]`
 - o `int age`
2. **Ask the user** how many contacts they want to add.
3. **Dynamically allocate** an array of `Contact` structures using `malloc()`.
4. **Prompt the user** to enter information for each contact.
5. **Display all contacts** using a loop.
6. **Free the memory** when you're done.

Bonus Ideas:

- Allow the user to search for a contact by name.
- Add an option to update or delete a contact.
- Save the contacts to a file for later use.

This project is a great way to practice combining multiple C concepts: structures, pointers, dynamic memory, and loops — all while making something practical!

Conclusion

Structures are an essential feature of C that allow developers to group variables of different types under a single name, enabling better data modeling and code

organization. In this part of the tutorial, you've learned how to declare and use structures, manipulate them with pointers, allocate memory dynamically, and manage collections using arrays of structures.

These techniques are crucial for writing organized, reusable, and efficient C code. By understanding when and why to use structures, you begin to think in terms of higher-level program design — a key step in becoming a proficient programmer.

Coming Up in The Next Episodes

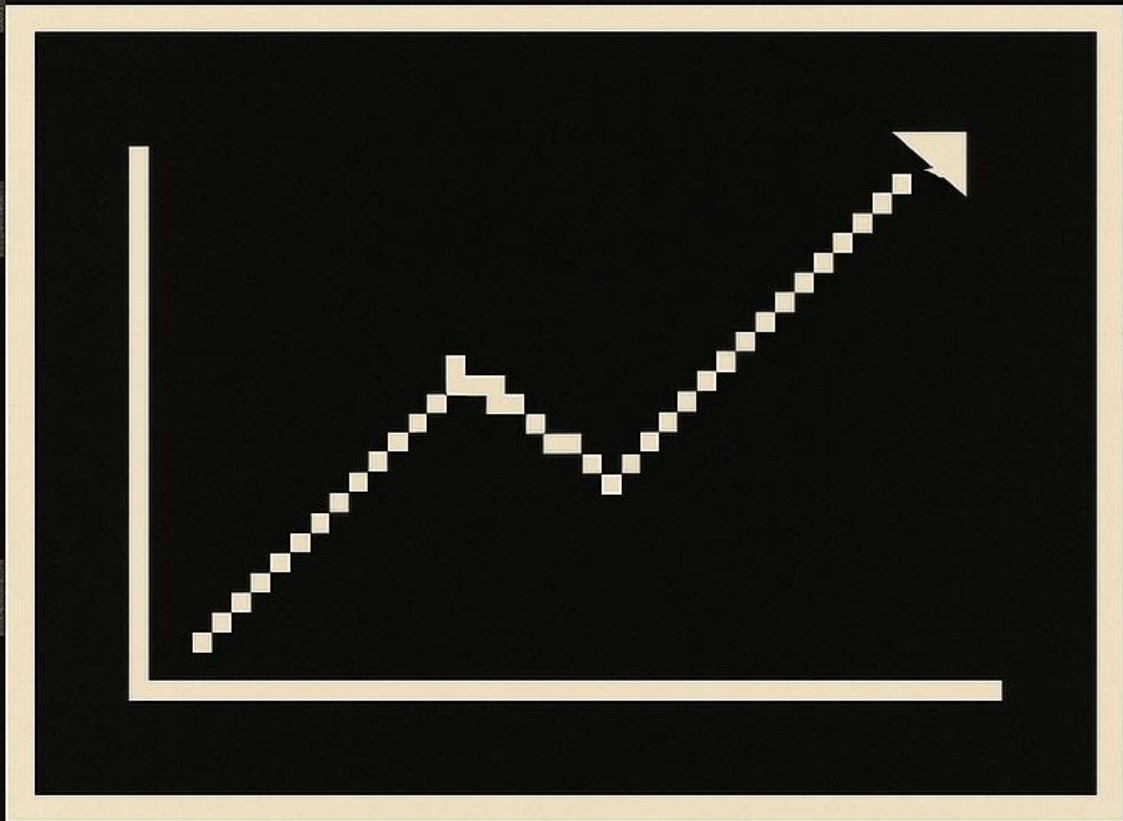
In the next installment of this C tutorial series, we'll dive deeper into more advanced topics that build upon your current knowledge. Here's a preview of what's to come:

- **File I/O in C** – Learn how to read from and write to files using standard C library functions. This allows your programs to store and retrieve data between runs.
- **Function Pointers and Callbacks** – Discover how to pass functions as arguments and use callbacks for flexible program design.
- **Data Structures in C** – Explore how to build linked lists, stacks, and queues — the foundation of many real-world software systems.
- **Preprocessor Directives and Macros** – Understand how the compiler processes your code before compilation, and how you can write powerful macros to generate code.
- **Modular Programming and Header Files** – Break your code into multiple source files, use header files effectively, and manage complex projects with clean interfaces.

These topics will help you transition from writing basic programs to developing efficient, real-world applications in C, but till then stay tuned and Happy Coding!



PROGRAMMING THE VGA CARD



DISPLAY GRAPHICS
IN TEXT MODE

*I*n our introductory episode to the wonders of text mode programming we have seen how easy ... or difficult can be to draw an ASCII table on the screen. The current iteration of the series will present you how to draw graphics on the text mode... or the closest approximation to what graphics can be in text mode.

Introduction to VGA Text Mode Fonts

The **VGA card** (Video Graphics Array), introduced by IBM in 1987, became the dominant standard for PC video hardware, setting a compatibility foundation that would last well into the future. Though later technologies like SVGA and beyond would build on it, basic VGA compatibility remained important for operating systems, BIOS programs, and even early graphical environments.

One of the most basic capabilities of the VGA was its **text mode**. In text mode, the screen is treated not as an array of pixels, but as a grid of character cells. Each cell corresponds to a character code combined with a color attribute, usually requiring two bytes of memory per cell. In the standard 80×25 text mode, there are 2000 character-cells, consuming about 4 KB of video memory. This screen memory is typically mapped starting at physical address **0xB8000** for color displays and **0xB0000** for monochrome ones.

Internally, the VGA does not render characters directly from their codes. Instead, it looks up each character code in a font table, a collection of small bitmap images called glyphs. Each glyph describes how the corresponding character should be drawn on the screen. These glyphs are typically 8 pixels wide and have a height of 8, 14, or 16 pixels depending on the text mode in use. The combination of character codes, attribute bytes, and glyph data allows the VGA hardware to efficiently generate a visual display from a relatively small amount of memory.

The font data itself is stored separately from the text buffer. On VGA hardware, fonts are usually located in video memory near the segment **0xA0000**, although how accessible or modifiable this memory is, depends on the current mode and the way the VGA's internal registers are

programmed. Each glyph for an 8×16 font, for example, requires 16 bytes of data (one byte for each scanline), meaning the full set of 256 characters occupies about 4 KB — a manageable size for the hardware of the era.

One of the remarkable features of VGA is that it allows programs to redefine these fonts dynamically. A program can upload its own font, effectively changing the entire appearance of the text screen. To assist with this, the system BIOS provides a convenient set of services accessible through interrupt **0x10h** (INT 10h), the standard video services interrupt. Among its many functions, INT 10h offers several subfunctions specifically for loading user-defined fonts.

The most common way to load a custom 8×16 font is to set the **AX** register to **0x1100h** and call INT 10h. When doing so, the program must also set up the **ES:BP** registers to point to the font data in memory, use **CX** to specify the number of characters to load (typically 256), and clear **DX** to zero to start from character code 0. The **BL** register specifies the font block size, where zero indicates an 8×16 font. After this interrupt call, the VGA card uses the newly uploaded font data for all character rendering on the screen.

In addition to this, there are other closely related subfunctions: setting **AX** to **0x1101h** loads an 8×14 font, while **AX** = **0x1102h** loads an 8×8 font. Each of these variants exists to match the different possible text mode scanline configurations that the VGA can operate under. For example, an 80×50 text mode would use smaller characters (typically 8×8) to fit more rows on the screen.

More advanced programs sometimes avoid the BIOS altogether and manipulate VGA font memory directly. This approach offers faster performance and greater flexibility, at the cost of requiring deeper knowledge of the VGA's internal registers. In particular, manipulating the VGA Sequencer and Graphics Controller registers allows the programmer to map font memory into the CPU's addressable memory space temporarily. Once mapped, the font glyphs can be written directly into memory, greatly speeding up the upload of large custom font sets. However, this method requires careful handling to avoid disrupting the VGA's operation, and is less portable than using the BIOS.

Historically, it is important to recognize that VGA evolved from the earlier EGA (Enhanced Graphics Adapter). While EGA also allowed user-defined fonts, it typically operated with 8×14 character glyphs by default rather than VGA's more common 8×16 glyphs. Although many of the INT 10h functions exist on EGA hardware, their behavior can be subtly different, and early EGA BIOSes sometimes lacked full support for all font-related services.

In practice, loading a new font involves either calling INT 10h with the appropriate parameters or directly manipulating memory if more control is needed. Programs that want to maximize compatibility often detect whether they are running on VGA or EGA hardware and select the correct font size and BIOS function accordingly. A simple check using INT 10h function AX = 0x1A00h allows a program to verify whether it is on a true VGA device; if not, it can assume EGA compatibility and adjust its behavior.

Overall, the VGA's handling of text mode fonts reflects its broader design philosophy: offering

powerful low-level capabilities while maintaining backward compatibility with older standards. Even today, understanding how the VGA manages text mode fonts provides valuable insights into early PC graphics programming and hardware design.

Before we get our hands dirty ... i.e. use some arcane C code to put a graphics in text mode on the screen, we need the graphics itself, so for the moment let's use the amazing Logo found here on the center of the page.

In order to have this graphics placed on the text screen, we will need to perform the basic

operation of breaking up the graphics into 8x16 glyphs, because that is what the VGA card can work with.

Since right now it is 2025, we can use a retro programming language that was released 20th. of February, 1991. Yes, we are talking about Python, which

is a programming language older than some programmers around. Here is a quick and dirty Python code that will perform this breaking up for us.

The following listing contains the source code of the python application you need to break up an image into smaller tiles. To run it install python on your favorite operating system.



```

from PIL import Image
import numpy as np
# Load and convert to grayscale
img = Image.open("image.png").convert("L")
img_data = np.array(img)
# Using standard VGA font size
tile_w, tile_h = 8, 16
tiles_x = img.width // tile_w
tiles_y = img.height // tile_h
unique_tiles = []
tile_indices = {} # mapping: tile as tuple -> index
tilemap = []
# A function to return the given tile from the image
def extract_tile(x, y):
    tile = img_data[y*tile_h:(y+1)*tile_h, x*tile_w:(x+1)*tile_w]
    return tuple(tuple(1 if pixel < 128 else 0 for pixel in row) for row in tile)
# Here the program starts
for y in range(tiles_y):
    row = []
    for x in range(tiles_x):
        tile = extract_tile(x, y)
        if tile not in tile_indices:
            tile_indices[tile] = len(unique_tiles)
            unique_tiles.append(tile)
        row.append(tile_indices[tile])
    tilemap.append(row)
# Tiles
for i, tile in enumerate(unique_tiles):
    print(f'const uint8_t tile{i}[{tile_h}][{tile_w}] = {{')
    for row in tile:
        row_str = ', '.join(str(pixel) for pixel in row)
        print(f'    {{ {row_str} }},')
    print('};\n')
# tile_set[]
print('const uint8_t* tile_set[] = {')
for i in range(len(unique_tiles)):
    print(f'    (const uint8_t*)tile{i},')
print('};\n')
# tilemap[][]
print(f'const uint8_t tilemap[{tiles_y}][{tiles_x}] = {{')
for row in tilemap:
    row_str = ', '.join(f'{idx}' for idx in row)
    print(f'    {{ {row_str} }},')
print('};')
# generic definitions
print(f'#define TILE_W {tile_w}\n')
print(f'#define TILE_H {tile_h}\n')
print(f'#define IMAGE_TILES_X {tiles_x}\n')
print(f'#define IMAGE_TILES_Y {tiles_y}\n')
print(f'#define TILES_COUNT {len(unique_tiles)}\n')

```


And since we can't expect everyone to be a master of reptiles, a quick overview of what we have here and how it is to be interpreted.

```
from PIL import Image
import numpy as np
```

The Python program begins by **importing two important libraries**: PIL (the Python Imaging Library), which allows us to load and manipulate images easily, and numpy, which is great for working with numbers in a matrix format. We'll need both, because we want to open a picture, cut it into pieces, and treat each piece as numbers.

```
img =
Image.open("image.png").convert("L")
img_data = np.array(img)
```

The first real step is loading the image. We open "image.png" and immediately **convert it to grayscale** using `.convert("L")`. This simplifies the problem because now every pixel is just a single number between 0 (black) and 255 (white), no colors involved. We then turn the image into a numpy array, so we can easily slice it up and process the pixels.

```
tile_w, tile_h = 8, 16
tiles_x = img.width // tile_w
tiles_y = img.height // tile_h
```

Next, we define the **size of a single tile**: 8 pixels wide and 16 pixels high, which matches the classic VGA text mode font size on DOS computers. Using integer division, we calculate how many tiles fit horizontally (`tiles_x`) and vertically (`tiles_y`) inside the image.

```
unique_tiles = []
tile_indices = {} # mapping: tile as
tuple -> index
tilemap = []
```

After setting up the basic image properties, the program prepares three data structures. It initializes an empty list called `unique_tiles` that will store only the different tiles we find. There's also `tile_indices`, a dictionary we use to quickly check if we have already seen a tile, and `tilemap`, which will eventually describe how to

rebuild the image using the tiles extracted from the image.

```
# A function to return the given tile
from the image
def extract_tile(x, y):
    tile =
img_data[y*tile_h:(y+1)*tile_h,
x*tile_w:(x+1)*tile_w]
    return tuple(tuple(1 if pixel < 128
else 0 for pixel in row) for row in
tile)
```

Before we start scanning the image, we define a helper function called `extract_tile(x, y)`. Given the coordinates of a tile, it **extracts a small rectangle** from the big image. Each pixel is then turned into either a 1 or a 0 — we treat any pixel darker than 128 as "black" (1), and everything else as "white" (0). The function returns the tile as a tuple of tuples, which is important because we need the tile to be a hashable object to use it as a key in a dictionary when identifying the unique tiles we need to write.

```
for y in range(tiles_y):
    row = []
    for x in range(tiles_x):
        tile = extract_tile(x, y)
        if tile not in tile_indices:
            tile_indices[tile] =
len(unique_tiles)
            unique_tiles.append(tile)
            row.append(tile_indices[tile])
    tilemap.append(row)
```

Now the real work begins. The program goes through the image tile-by-tile, line-by-line. For each tile, it extracts its pixel data and checks if it has already been seen. If it's new, it is added to the `unique_tiles` list, and its index is recorded in `tile_indices`. Regardless of whether it's new or already known, the tile's index is added to the current row of the `tilemap`, because the tile will participate in the reconstruction of the original image. After finishing each row, the row is added to the `tilemap` grid.

Once the image is fully processed, the program **prints the C code** that defines each unique tile as a 2D array of bytes (`uint8_t`). Each tile gets its own array, named something like `tile0`, `tile1`,

and so on. As you easily have guessed, these tiles will be the ones that we use to redefine the font of the VGA card, to display our graphics.

After printing the tiles themselves, the program creates a C array called `tile_set`, which is simply a list of pointers to all the tile arrays. This will make it easy for a C program to access any tile by its index later.

Next, the program prints out the `tilemap`, which is a two-dimensional array where each number tells which tile goes at a particular (x, y) position. In other words, this is the recipe for reconstructing the original image from the tiles.

Finally, the script prints out some useful `#define` constants for the C side: the width and height of a tile, the number of tiles horizontally and vertically, and the total number of unique tiles.

There is just one drawback to this entire process: if the image is too big, there will be more than 256 unique tiles, so we recommend that you will reduce the size of the image till you will get tiles that fit the number.

You can run the script from the terminal, for example if you have saved it as `imagetiler.py` then simply executing it and piping the output into a file will deliver you the desired result.

Feel free to redirect the output of this script to a file called `"bits.c"` because that is the one used in the VGA font overwriter.

When the script is done executing, its output will be something like:

```
#include <dos.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <malloc.h>
#include <conio.h>

typedef unsigned char uint8_t;
typedef unsigned int uint16_t;

#include "bits.c"
```

```
const uint8_t tile1[16][8] = {
    { 0, 0, 0, 0, 0, 0, 0, 0 },
    { 0, 0, 0, 0, 0, 0, 0, 0 },
    { 0, 0, 0, 0, 0, 0, 0, 0 },
    { 1, 0, 0, 0, 0, 0, 0, 0 },
    { 1, 1, 1, 1, 1, 0, 0, 0 },
    { 1, 1, 1, 1, 1, 0, 0, 0 },
    { 1, 1, 1, 1, 1, 0, 0, 0 },
    { 1, 1, 1, 1, 1, 1, 0, 0 },
    { 1, 1, 1, 1, 1, 1, 0, 0 },
    { 1, 1, 1, 1, 1, 1, 1, 0 },
    { 1, 1, 1, 1, 1, 1, 1, 0 },
    { 0, 1, 1, 1, 1, 1, 1, 1 },
    { 0, 1, 1, 1, 1, 1, 1, 1 },
    { 0, 1, 1, 1, 1, 1, 1, 1 },
    { 0, 1, 1, 1, 1, 1, 1, 1 },
    { 0, 1, 1, 1, 1, 1, 1, 1 },
};

const uint8_t* tile_set[] = {
    (const uint8_t*)tile0,
    (const uint8_t*)tile1,

const uint8_t tilemap[15][40] = {
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
#define TILE_W 8
#define TILE_H 16
#define IMAGE_TILES_X 40
#define IMAGE_TILES_Y 15
#define TILES_COUNT 219
```

We have omitted the rest of the 218 tiles, to conserve space.

And since we were talking about it, it is time that we present the actual program that does the overwriting of the fonts in order to display the graphics. In text mode. The program is in the listing below, and as expected, it will be followed by a very thorough presentation.


```

void write_font_char(const uint8_t * tile, int index, uint8_t * fontMem) {
    for (int y = 0; y < TILE_H; y++) {
        uint8_t byte = 0;
        for (int bit = 0; bit < TILE_W; bit++) {
            if (tile[y * TILE_W + bit]) {
                byte |= (0x80 >> bit);
            }
        }
        *((uint8_t * ) fontMem + 16 * index + y) = byte;
    }
}

void enable_custom_font(unsigned int segment, unsigned int ofs) {
    _asm {
        mov ax, 0x03          // 0x03 - Text Mode
        int 0x10              // Set it
        mov ax, 0x1114        // Specify 8x16 Fonts
        xor bx, bx
        int 0x10
        mov ax, segment        // Segment of font memory location
        mov es, ax             // Goes into ES
        mov ax, ofs            // Offset of font memory location
        mov bp, ax             // ES:BP -> font table location
        mov ax, 0x1110         // Function to load user-defined character generator
        mov bh, 16             // height
        mov bl, 0              // Font block 0
        xor dx, dx             // Starting index 0
        mov cx, TILES_COUNT    // Number of characters to load (up to 256) TILE_COUNT 218
        int 0x10
    }
}

void draw_image() {
    uint16_t __far * screen = (uint16_t __far * ) MK_FP(0xB800, 0);
    for (int y = 0; y < 25; y++) {
        for (int x = 0; x < 80; x++) {
            if (y < IMAGE_TILES_Y && x < IMAGE_TILES_X) {
                const uint8_t tile_index = tilemap[y][x];
                screen[y * 80 + x] = 0x0700 | tile_index; // white on black
            } else {
                screen[y * 80 + x] = 0x0700; // white on black
            }
        }
    }
}

int main() {
    uint8_t * farPtr = (uint8_t * ) calloc(TILES_COUNT * 16, 1);

    // Upload all unique tiles into VGA font memory
    for (int i = 0; i < TILES_COUNT; i++) {
        write_font_char(tile_set[i], i, farPtr);
    }

    // Get the segment and offset of the far pointer

```

```

unsigned int segment = FP_SEG(farPtr);
unsigned int offset = FP_OFF(farPtr);

enable_custom_font(segment, offset);

draw_image();

getch();

_asm {
    mov ax, 0x3
    int 0x10
}
free(farPtr);
return 0;
}

```

And, as promised the explanation. Like every well behaving C file, this also starts with a set of standard includes.

```

#include <dos.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <malloc.h>
#include <conio.h>

```

- **dos.h**: Provides access to DOS-specific functions and data structures. Specifically, it often contains macros like **MK_FP**, **FP_SEG**, and **FP_OFF** for working with segmented memory.
- **stdio.h**: Standard input/output functions (e.g., **printf**, **scanf**).
- **string.h**: String manipulation functions (e.g., **strcpy**, **strlen**).
- **stdlib.h**: General utility functions, including memory allocation (**calloc**, **free**) and conversions.
- **malloc.h**: Another header for memory allocation functions. In many DOS environments, it's largely equivalent to **stdlib.h**.
- **conio.h**: Console input/output functions, providing functions like **getch()** (get character from the console without echoing).

The next two lines were introduced for making possible the compilation of the program using TurboC++ since at some point during development we ran into some issues with Watcom C++ and DosBox, but more about these later.

```

typedef unsigned char uint8_t;
typedef unsigned int uint16_t;

```

The two following types are defined for TurboC++ which does not have these out of the box. If you decide to compile the source file with Watcom compiler, then you might not need these explicit definitions, Watcom comes out of the box with them.

- **uint8_t**: Defines an unsigned 8-bit integer type (a byte).
- **uint16_t**: Defines an unsigned 16-bit integer type (a word).


```
#include "bits.c"
```

This line includes the contents of the file `bits.c` directly into the current source file during compilation, as we have presented it in the section detailing the generation of the tiles. The `bits.c` file contains the following definitions:

- `TILE_W`: The width of each tile in pixels (8, the standard width of a VGA text-mode character).
- `TILE_H`: The height of each tile in pixels (16, the height VGA for text-mode characters).
- `TILES_COUNT`: The total number of unique tiles used to compose the image.
- `tile_set`: A 2D array (or an array of pointers) that holds the pixel data for each tile.
- `IMAGE_TILES_X`: The width of the image in tiles.
- `IMAGE_TILES_Y`: The height of the image in tiles.
- `tilemap`: A 2D array that maps tiles to screen positions. As presented already, it is structured as `tilemap[IMAGE_TILES_Y][IMAGE_TILES_X]`. Each element in `tilemap` is an index into the `tile_set` array, indicating which tile to display at that location on the screen.

```
void write_font_char(const uint8_t * tile, int index, uint8_t * fontMem) {
    for (int y = 0; y < TILE_H; y++) {
        uint8_t byte = 0;
        for (int bit = 0; bit < TILE_W; bit++) {
            if (tile[y * TILE_W + bit]) {
                byte |= (0x80 >> bit);
            }
        }
        *((uint8_t *) fontMem + 16 * index + y) = byte;
    }
}
```

This function takes the pixel data of a single tile and writes it to the font memory. The parameters are:

- `tile`: A pointer to the pixel data of the tile (from the `tile_set` array).
- `index`: The index of the tile within the character set (0 to `TILES_COUNT - 1`).
- `fontMem`: A pointer to the memory location where the custom font is being built.

The operation of the function is as follows:

- **Outer Loop**: Iterates through each row of the tile (0 to `TILE_H - 1`).
- **Inner Loop**: Iterates through each bit (pixel) in the current row (0 to `TILE_W - 1`).
- `if (tile[y * TILE_W + bit])`: Checks if the pixel at the current position is set (1). The expression `y * TILE_W + bit` calculates the linear offset of the pixel within the tile's data.
- `byte |= (0x80 >> bit) ;`: If the pixel is set, this line sets the corresponding bit in the byte variable.
 - `0x80` is the binary value 10000000.
 - `0x80 >> bit` shifts the 1 bit to the right by `bit` positions. For example, if `bit` is 0, the result is 10000000. If `bit` is 1, the result is 01000000, and so on.

- The `|=` operator performs a bitwise OR, setting the appropriate bit in the byte.
- `*((uint8_t*)fontMem + 16 * index + y) = byte;` This line writes the calculated byte (representing one row of the tile) to the correct location in `fontMem`.
 - `fontMem + 16 * index`: Calculates the starting address in `fontMem` for the character with the given index. Each character's font data occupies 16 bytes (one byte per row for a 16-pixel high character).
 - `+ y`: Adds the offset for the current row.
 - `*((uint8_t*) ...) = byte;` The `(uint8_t*)` cast ensures that the pointer arithmetic is done in bytes, and the `*` dereferences the pointer to write the byte value.

The next function is pure assembly magic, and thankfully is full with comments too.

```
void enable_custom_font(unsigned int segment, unsigned int ofs) {
    _asm {
        mov ax, 0x03          // 0x03 - Text Mode
        int 0x10              // Set it
        mov ax, 0x1114        // Specify 8x16 Fonts
        xor bx, bx
        int 0x10
        mov ax, segment        // Segment of font memory location
        mov es, ax             // Goes into ES
        mov ax, ofs            // Offset of font memory location
        mov bp, ax             // ES:BP -> font table location
        mov ax, 0x1110         // Function to load user-defined character generator
        mov bh, 16             // height
        mov bl, 0              // Font block 0
        xor dx, dx             // Starting index 0
        mov cx, TILES_COUNT    // Number of characters to load (up to 256) TILES_COUNT 218
        int 0x10
    }
}
```

This function uses the assembly instruction to interact with the VGA BIOS and load the custom font:

- **segment**: The segment address of the memory where the custom font data is stored.
- **ofs**: The offset address of the font data.
- **_asm { ... }**: Indicates an inline assembly block.
- **mov ax, 0x03; int 0x10**: Sets the video mode to 80x25 color text mode.
 - **mov ax, 0x03**: Loads the value 0x03 into the **AX** register.
 - **int 0x10**: Calls the VGA BIOS interrupt. 0x10 is the main video BIOS interrupt. **ax** holds the function number. 0x03 is the function to set the video mode.
- **mov ax, 0x1114; xor bx, bx; int 0x10**: Redefines the characters on VGA cards to use the standard 16-scanline-high text-mode font, and it takes the additional steps needed to activate the font.
 - **mov ax, 0x1114**: Loads 0x1114 into **AX**. As explained above, this is a specific value for `int 0x10`, it will instruct the VGA card to use the specific 8x16 font.

- `xor bx, bx`: Sets the **BX** register to 0. **BL** can be used to select a specific character block, we use block 0.
- `int 0x10`: Calls the BIOS interrupt.
- `mov ax, segment; mov es, ax`: Sets the **ES** (Extra Segment) register to the segment address of the font data. The VGA BIOS often expects the font data to be pointed to by **ES:BP**. We need this double move operation, since older processor do not allow moving a variable into a segment register.
- `mov ax, ofs; mov bp, ax`: Sets the **BP** (Base Pointer) register to the offset address of the font data. Now, **ES:BP** points to the beginning of the custom font data. Again, please observe the double move to initialize the register.
- `mov ax, 0x1110`: Loads 0x1110 into **AX**. This selects the BIOS function to load a user-defined character generator.
- `mov bh, 16`: Sets **BH** to 16, indicating that the character height is 16 pixels.
- `mov bl, 0`: Sets **BL** to 0, indicating the starting character block (usually the first 256 characters).
- `xor dx, dx`: Sets **DX** to 0, indicating the starting character code to redefine (starting from character code 0).
- `mov cx, TILES_COUNT`: Loads the number of characters to redefine into **CX**.
- `int 0x10`: Calls the BIOS interrupt to load the custom font.

The next function, as the name suggests draws the image, as expected.

```
void draw_image() {
    uint16_t __far * screen = (uint16_t __far * ) MK_FP(0xB800, 0);
    for (int y = 0; y < 25; y++) {
        for (int x = 0; x < 80; x++) {
            if (y < IMAGE_TILES_Y && x < IMAGE_TILES_X) {
                const uint8_t tile_index = tilemap[y][x];
                screen[y * 80 + x] = 0x0700 | tile_index; // white on black
            } else {
                screen[y * 80 + x] = 0x0700; // white on black
            }
        }
    }
}
```

The function writes to the VGA text mode screen buffer to display the image in a way we have presented in our previous episode of the series:

- `uint16_t __far* screen = (uint16_t __far*)MK_FP(0xB800, 0);`: Creates a far pointer to the beginning of the VGA text mode video memory.
 - **0xB800**: The segment address of the color text mode buffer.
 - **0**: The offset within that segment.
 - **MK_FP**: A macro (from `dos.h`) that combines a segment and offset into a far pointer.

- `uint16_t __far*`: Declares screen as a far pointer to 16-bit words. Each character cell in the text buffer is 2 bytes: one for the character code, and one for the attribute (color, etc.).
- **Outer Loop**: Iterates through the rows of the text mode screen (0 to 24).
- **Inner Loop**: Iterates through the columns of the text mode screen (0 to 79).
- `if (y < IMAGE_TILES_Y && x < IMAGE_TILES_X)`: Checks if the current screen position is within the bounds of the image.
- `const uint8_t tile_index = tilemap[y][x];`: Retrieves the index of the tile to display from the tilemap array.
- `screen[y * 80 + x] = 0x0700 | tile_index;`: Writes the character code and attribute to the screen buffer.
 - `y * 80 + x`: Calculates the offset of the character cell in the linear screen buffer.
 - `tile_index`: The character code. Since we've redefined the character set, this index will display the corresponding custom tile.
 - `0x0700`: The attribute byte. `0x07` represents white text on a black background. The attribute is placed in the *high* byte of the 16-bit word.
 - `|`: The bitwise OR operator combines the attribute byte and the tile index.
- `else { screen[y * 80 + x] = 0x0700; }`: If the screen position is outside the image bounds, it writes a space character (character code 0) with white on black attributes. Because we've reloaded the font, character 0 will now display our first custom tile, which for our convenience is an empty tile.

What remains is the main function:

```
int main() {
    uint8_t * farPtr = (uint8_t * ) calloc(TILES_COUNT * 16, 1);
    for (int i = 0; i < TILES_COUNT; i++) {
        write_font_char(tile_set[i], i, farPtr);
    }
    unsigned int segment = FP_SEG(farPtr);
    unsigned int offset = FP_OFF(farPtr);
    enable_custom_font(segment, offset);
    draw_image();
    getch();
    _asm {
        mov ax, 0x3
        int 0x10
    }
    free(farPtr);
    return 0;
}
```

The following is done in the main function:

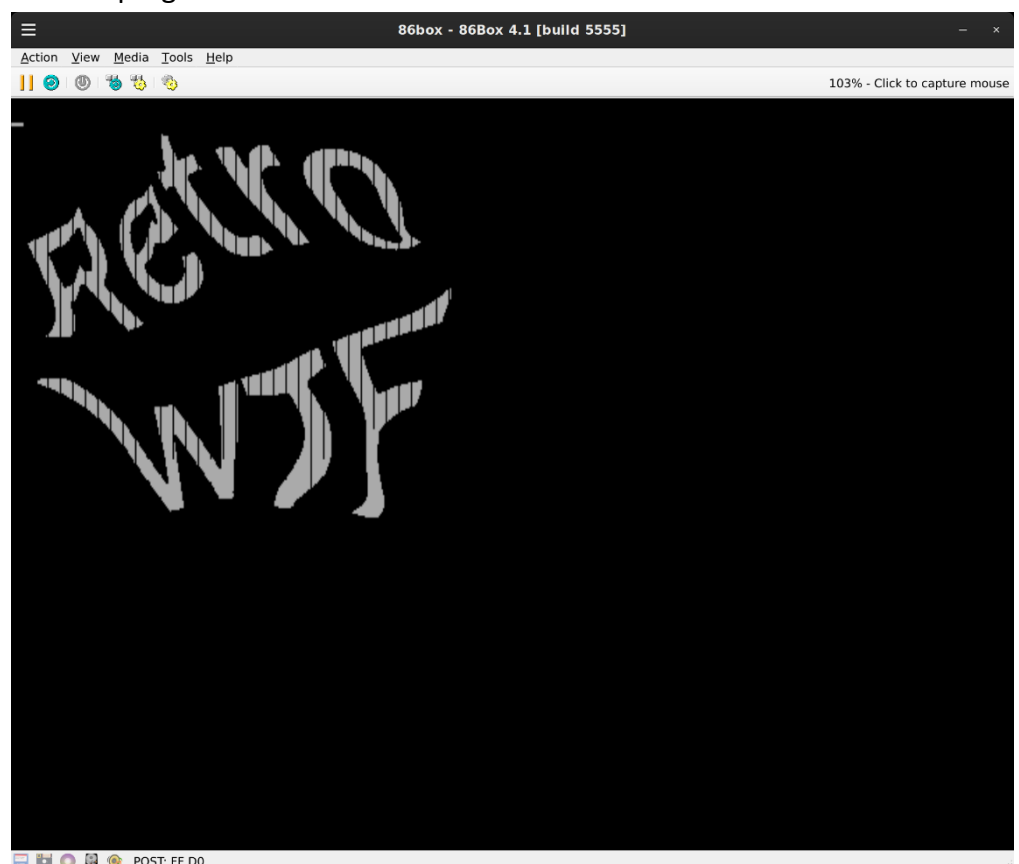
- `uint8_t *farPtr = (uint8_t*)calloc(TILES_COUNT * 16, 1);`: Allocates memory to store the custom font data.

- `calloc`: Allocates memory and initializes it to zero.
- `TILES_COUNT * 16`: Calculates the total number of bytes needed (number of tiles times 16 bytes per tile).
- `1`: The size of each element (1 byte).
- `uint8_t *farPtr`: Declares a far pointer to the allocated memory. In DOS, far pointers are needed to access memory outside the default data segment.
- **Font Data Loop**: The `for` loop calls `write_font_char` for each tile in `tile_set` to copy the tile's pixel data into the allocated memory pointed to by `farPtr`.
- `unsigned int segment = FP_SEG(farPtr);`: Gets the segment address of the allocated memory using the `FP_SEG` macro.
- `unsigned int offset = FP_OFF(farPtr);`: Gets the offset address of the allocated memory using the `FP_OFF` macro.
- `enable_custom_font(segment, offset);`: Calls the function to load the custom font into the VGA BIOS.
- `draw_image();`: Calls the function to write the tile indices to the screen buffer, displaying the image.
- `getch();`: Waits for a key press before exiting. This keeps the image on the screen until the user is ready to close the program.
- `_asm { mov ax, 0x3; int 0x10; }`: Resets the video mode to 80x25 color text mode before the program exits. This restores the original text mode.
- `return 0;`: Indicates successful program execution.

When all that above is in place, compiles, links and runs as expected, the following output will be shown on the screen.

Now, certainly, you may ask what is with those ugly vertical black lines, and rightfully the question comes: why? The answer is a bit more complicated than obvious.

When you work with text mode on VGA, every character you see on the screen is built from a small grid of pixels. For normal VGA



fonts, this grid is usually **8 pixels wide and 16 pixels tall**. Each character, like 'A', 'B', or a box-drawing symbol, is stored as **16 bytes**, where **each byte represents one horizontal line** of the character.

Inside each byte, **the 8 bits represent 8 pixels**. The **highest-order bit (bit 7)** is the **leftmost pixel**, and **bit 0** is the **rightmost pixel** of the 8. So the actual font data only defines an **8-pixel-wide shape**.

However, the **physical screen** in VGA text mode is actually **720 pixels wide** for an 80×25 screen, **not just 640 pixels** like older EGA graphics modes. **$720 \div 80 \text{ columns} = 9 \text{ pixels per character}$** .

This means that when the VGA card displays a character, it *shows 9 pixels across*, **even though the font only defines 8**.

Now here's where the interesting trick happens:

- For **most characters**, the **9th pixel** (the extra one) is just **blank** (off). This makes reading the fonts in text mode easier with that tiny space between them.
- But for **special characters** — like the box-drawing symbols (`␣` to `␣` in the character set) — VGA **automatically copies** the **8th bit** of the font and uses it to **fill the 9th pixel**.

Why? Because if the right edge of the character was blank, you would see ugly "gaps" when drawing borders or tables.

Imagine you're drawing a box, and if the rightmost line suddenly disappears, the box would look broken!

In other words:

- For box characters (`␣` to `␣`), VGA automatically **extends** the rightmost pixel (bit 0) to the invisible 9th pixel.
- For normal characters, the 9th pixel stays empty.

This behavior is **built into VGA hardware**. You don't normally control it unless you go into **special VGA registers** to disable it - but most programs leave it on because it makes everything look correct.

And as a last interesting side note: The program does not run in DosBox. Regardless its flavor (vanilla, x, staging ...), it does not want to interact properly with our tiny little program. Either that we have some undetected access violation in our program DosBox does not like, or that it seems that we have reached the limits of emulation of what DosBox can offer.

Anyway, we have 86Box, or the hardcore of us have their physical retro machines, so there is no need to worry.

Why Understanding VGA Font Loading Still Matters Today

Although the world has largely moved on from VGA hardware, the knowledge of how text mode fonts are loaded, modified, and managed remains surprisingly relevant - and not just for historical interest.

Mastering these low-level techniques teaches lessons about computer architecture, memory mapping, hardware control, and software efficiency that are timeless in their value.

For one, retrocomputing is a vibrant and growing field. Hobbyists restoring vintage PCs, writing demos for old hardware, or developing new DOS-based games and utilities need to understand the real mechanisms behind VGA behavior to create authentic experiences. Without understanding how the VGA exposes and

protects font memory, it would be impossible to correctly replicate the techniques that made early software so responsive and visually distinctive.

Additionally, this knowledge forms a critical bridge to understanding modern graphics hardware. While today's GPUs are vastly more complex, the basic ideas of planes, memory mapping, register programming, and controlling hidden layers of data are still alive. Many concepts used in manipulating the VGA's memory layout - such as bank switching, special-purpose memory areas, and hardware-accelerated copying - echo in modern systems, albeit at a much greater scale.

Beyond that, low-level programming skills like these build a mental model of how computers actually function beneath all the abstraction layers. For operating system development, embedded systems, or security research, the ability to think in terms of direct hardware access is an essential tool. Knowing how the VGA hardware separates different memory planes, how it masks writes and controls access permissions, mirrors modern techniques for memory protection, direct memory access (DMA), and hardware virtualization.

There is also an artistic side to this mastery. The elegance with which developers once bent limited hardware to their will - achieving smooth animations, clever UI tricks, and rich text-mode graphics - represents a kind of craftsmanship that is still admired today. Many of the graphical effects that defined early PC gaming and software were only possible because programmers understood and exploited the fine details of font management and screen rendering.

In a way, studying low level system programming is like studying classical architecture: the technologies may be ancient, but the ingenuity, creativity, and discipline they teach are eternally valuable.

Thus, even in an era of 4K displays and real-time 3D graphics, the lessons of VGA font handling continue to inspire and inform those who seek to understand computers at their deepest level.

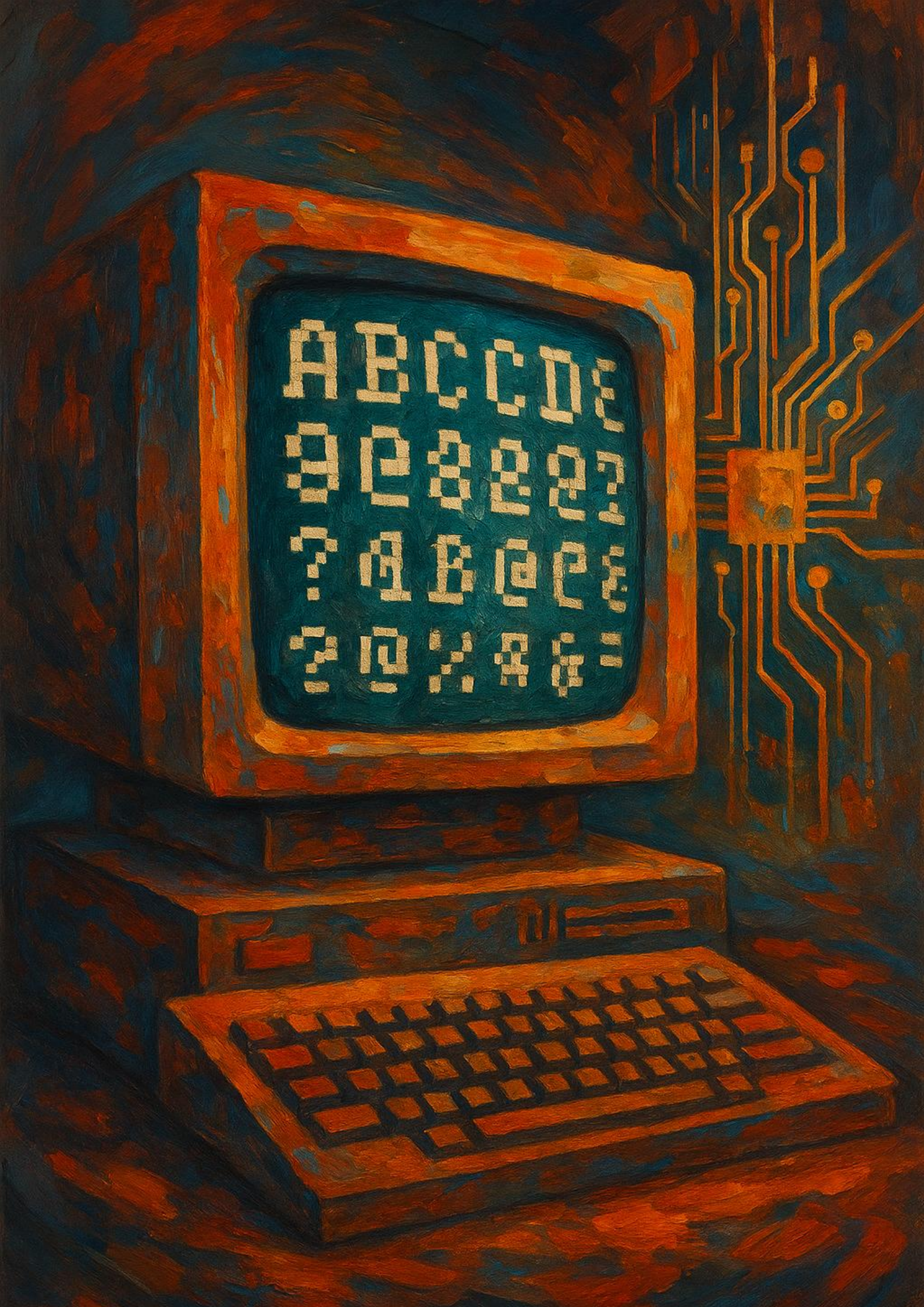
What the future brings?

Indeed, we have had an interesting chapter today on low level VGA font handling. But in our next episode we will take things even further.

We will try to do some text mode animation while still remaining in the domain of VGA cards, text mode programming and obviously our beloved fonts too.

Because without fonts life has not taste.





The Story of How Have I Been Hacked

— and subsequently lost my faith in one
paid operating systems' provider —



“ The long, tragicomic saga of how I got hacked - and lost my faith in a certain paid operating system vendor - could have ended on a brighter note, but apparently, the universe had other plans...

During the final week of July, on a pleasantly sunny afternoon, something rather disconcerting occurred just as I was putting the finishing touches on my latest article. I was comfortably settled in my armchair, mulling over how to rephrase an awkwardly worded sentence, when my phone gave a brief, insistent buzz.

Like any other journalist hopelessly tethered to their devices, I reached for it immediately. To my astonishment, I found a notification from my Microsoft Authenticator app informing me that the credentials for my account had been updated. Or, to put it plainly: it appeared I had just changed the password to my Microsoft account.

... Except that I didn't.

For the past 45 minutes, I had been sitting diligently in my reclining chair, attempting to coax the words of an article into conforming to the proper grammatical standards of the English language. The words resisted, though not with much success. Then, alarmed by a sudden notification, I did precisely what anyone in my situation would have done.

I attempted to log in to my Microsoft account using my old password. As an aside, I should mention that at that particular moment, I was working on an article about the Linux OS, using my Linux machine, which had rarely, if ever, been used to access Microsoft-related sites. To log in, I was prompted to enter both my ID and password again. And lo and behold, the password didn't work.

As I stared in disbelief at the screen, my phone vibrated once more. With trembling hands, fearing the worst, I checked it again. There, glaring back at me, was another cold notification from the same Microsoft Authenticator app: my Microsoft account had been closed. Deleted, erased from existence, utterly obliterated.

All within the space of the five minutes since I'd first picked up the initial notification.

My brain immediately shifted into emergency mode—I could almost hear the tiny cogs whirring frantically. Then it dawned on me: my recovery email for the account was the one I used daily, so I quickly switched tabs to see if there had been any suspicious activity. It's worth noting that this recovery email was hosted on Google's platform. It's a private email I've used for about 20 years, ever since Gmail was launched, back when getting an account required an invitation from someone already on the platform.

Returning to my Google account, I found four unread emails from Microsoft. The first two informed me of suspicious activity on my account, urging me to review and secure it.

From my perspective, my account was quite secure. A 14-character password with a mix of upper and lower case letters, digits, and two different punctuation marks. And, crucially, this password was used solely for this account.

I also had two-factor authentication enabled, with the authenticator app exclusively on my phone and nowhere else.

The third email detailed how my authentication credentials had been changed by an IP address whose endpoint was in Estonia. Now, I've visited Estonia several times—lovely country, beautiful medieval architecture in Tallinn, excellent food, and very friendly people. I don't recall ever irritating anyone there to the extent that they'd hack my account in

retaliation. So, it seemed likely that this IP address belonged to a hacker located somewhere entirely different.

The saddest of the four emails was the last one, which notified me that my Microsoft account had been deleted, and they thanked me for my business.

Then, something truly frightening happened. After a few minutes of stunned silence, during which I mourned the loss of my account, the four emails vanished from my inbox right before my eyes. I couldn't believe what I was witnessing—emails don't just disappear without human intervention.

I had no rule in place to automatically delete emails from Microsoft, especially those related to the safety and security of my accounts. I headed over to the Trash folder in my Gmail account, and sure enough, there were the four emails... and then they disappeared again. I watched in real-time, from the front row, as my account was being hacked right before my eyes.

I thought there was no higher state of mind than emergency mode, but it turns out there is: survival mode. My brain instantly switched to it when it realized that my account was under unauthorized access at that very moment. Not willing to risk losing all the history I'd accumulated over the past two decades, my first instinctive action was to immediately change the password of my account, which was, at that very moment, hosting uninvited guests.

This account, too, had a 15-character, unique and strong password, coupled with two-factor authentication powered by Google Authenticator. The password had been changed recently, just as a matter of routine. At that point, I hadn't identified any clear way someone could have gained access to it.

Despite these security measures, someone was freely roaming around my account, able to do whatever they pleased while cleverly covering their tracks. It seems that erasing my Microsoft account was their primary objective, or perhaps my timely password change interrupted their nefarious plans.

I spent the following hours checking all my accounts for any signs of compromise, but thankfully, there were none. It appears the only casualties were my main Gmail account and the associated Microsoft account. Nevertheless, to remain on the safer side of the internet—if such a thing exists—I changed the passwords to all accounts that have any meaningful connection to my real life.

The recovery plan

With a clearer head, it was time to devise a recovery plan. Since my account had vanished, the first thing I did was attempt to recreate it, and to my utter amazement, I succeeded—using the same old email address. My unspoken, deepest hope was that once the account was live again, everything I'd had stored there would miraculously return to me, or so my naive self believed.

Oh, how wrong I was.

What greeted me was a completely empty account. Not a single trace of what I had expected to find. It was as if the account had been freshly created. And, in truth, that's precisely what had happened. The worst of all was that I have also lost my access to my Xbox account, the main reason why in the first I actually have a Microsoft account: to play Minecraft.

It was time to escalate this matter to a higher level of urgency. I decided that the best course of action would be to contact Microsoft support and seek their assistance in recovering my account.

But before we delve into this rather serious issue, let's take a moment to enjoy the following joke, courtesy of <https://flunkingfamily.com/best-microsoft-joke-ever-will-enjoy/>:



A pilot is flying a small, single-engine, charter plane with a couple of really important executives on board into Seattle airport. There is fog so thick that visibility is 40 feet, and his instruments are out. He circles looking for a landmark and after an

hour, he is low on fuel and his passengers are very nervous.

At last, through a small opening in the fog he sees a tall building with one guy working alone on the fifth floor. Circling, the pilot banks and shouts through his open window: "Hey, where am I?"

The solitary office worker replies: "You're in an airplane."

The pilot immediately executes a swift 275 degree turn and executes a perfect blind landing on the airport's runway five miles away. Just as the plane stops, the engines cough and die from lack of fuel.

The stunned passengers ask the pilot how he did it.

"Elementary," replies the pilot, "I asked the guy in that building a simple question. The answer he gave me was 100% correct but absolutely useless; therefore, I knew that must be at Microsoft's support office and from there the airport is three minutes away on a course of 87 degrees".

I had some reservations about resorting to this procedure, as I'd been familiar with that joke for several years. However, given the level of my desperation, there was simply no other option left to me.

After spending countless hours over several days in live chats with various tiers of support engineers, I came to realize that some jokes might indeed have a seed of truth.

But before this starts sounding too heated, I must say this: every support engineer I actually spoke with was exceptionally professional, extremely helpful, and truly empathetic. They went to great lengths to try and resolve the problem I presented them with.

A huge thank you to all of you—you did a marvellous job within the limits and constraints of your roles, and I am genuinely grateful for all the assistance you provided.

But...



Despite your commendable efforts, which I admired, and your patience, which I envied, my issues remained unresolved. It seemed that once the escalated cases left your desks, they encountered an impenetrable corporate wall of policies—a collection of rules designed not for the customer's benefit but rather to maximize profit. After going around in circles multiple times, I got the distinct impression that my issue would never be resolved.

I wasn't asking for much: just that my purchases made with the previous account be transferred to my new one. The response I received was as follows:

"Unfortunately, due to security protocols and limitations, it is not possible to transfer an existing Office subscription and associated data to a different account. Each subscription is tied to a specific account,

and transferring it would compromise security and violate licensing agreements."

And another response:

"The only option we have is to permanently suspend this account to prevent any further use. At this time, I have successfully suspended this account, and this will remain on indefinitely."

In simpler terms, this meant that everything I had purchased with my previous account was lost forever. I would either have to accept the loss or repurchase everything, simply because I was too insignificant for a large company to go the extra mile.

The problem wasn't a technical one. In today's cloud-based world, I'm quite certain that all my data, along with its associated IDs, is sitting somewhere in a backup database. Crafting a clever SQL query to restore everything to its former state would likely be a 10-minute task for a knowledgeable support agent. The issue was more about the corporate mentality that, in order to have a successful and profitable company, every interaction must maximize revenue, often at the expense of doing what's right, all while hiding behind license agreements and rules that most customers don't understand.

This isn't the fault of the support engineers who tried to help me. They did everything they could and were very sympathetic to my situation, but their hands were tied by the aforementioned rules, regulations, and... well, that's just how it is.

However, there's something I must mention. During the so-called "resolution" of my case, a few rather obscure and inexplicable scenarios occurred, like lightning from a clear blue sky.

The first incident truly shocked me because it defied any logical explanation. At one stage in the process, I was asked to create a new, empty Microsoft account to serve as a contact point for customer service. In order to communicate with Xbox support (which is another story altogether), I also created an associated Xbox account, as this was where the major damage

was expected to be sorted out after the supposed “resolution” of my main account, which had led to its complete obliteration.

Two weeks into the process, just a few days after creating this account, the newly created account was closed, due to:

“The Xbox Safety team has found that recent behavior by the Xbox profile based on your email address violated one or more terms of the Community Standards for Xbox or Microsoft Services Agreement.

The violation was either brought to our attention by complaints from other players or discovered in the course of moderating the Xbox service. Members of the Xbox team have reviewed evidence of the violation and appropriate consequences. ”

For heaven’s sake, that account was never even used! I had logged into it once, when I have created it, no games, no activity, it was as fresh as a two-week-old baby, and yet it somehow violated some service agreement. No explanation, no comments, no clarity on what section was breached or what actions led to this. I was left utterly baffled. Imagine if this had been an active account, where people chat, play dozens of games, and have purchases linked to it, only to have it suspended without any explanation.

Fortunately, there was an “appeal” button. After clicking it, I was presented with a tiny text field, just enough space to type, “This is a new account created for the purpose of restoring the old one”. Someone must have actually read it, because a few days later, I received a message stating the suspension had been lifted. This was a relief, not that I had any chance to use the account by that point, but I certainly didn’t want to go through the whole process again.

The second oddity occurred about two and a half weeks into the process (which, according to Microsoft’s own people, should take no more than 24 to 72 hours). I received another email from the Microsoft Support Team requesting the following information:

- Microsoft Account ID
- Gamertag
- Console ID

Apparently, they couldn’t locate these details in the original request I had opened. I would have been happy to provide this information immediately... except the email came from a generic address with no way to reply directly.

I was momentarily stumped, and I attempted to contact Xbox support to resolve this issue. Navigating the site was straightforward enough, but finding someone who could actually provide assistance was anything but. I had to jump through several hoops with an automated bot that pretended to help, and if you’re lucky (meaning you’ve pressed all the right buttons, much like in a game), you might eventually get the chance to ask a question to a member of the gamer community.

After turning to the community for help, I once again encountered some truly kind and helpful gamers, and they quickly pointed out that the only way to provide the requested information to Microsoft Support was to reply to that generic email—they’d sort it out on their end somehow.

So, that’s exactly what I did. And wouldn’t you know it, they did resolve the issue with my Xbox account promptly. Just as expected, by suspending, disabling, and obliterating that too. Thanks, Microsoft. Here’s an excerpt from the email they sent me:

“A couple of things to note regarding the account suspension:

*If you use this account for **Minecraft**, we regret to inform you that the Minecraft portion of the account is also unable to be recovered and the game will need to be re-purchased on a new account. We understand that this is not the news that you wanted to hear and apologize for any inconvenience that this may cause.*

*In the event that you have files stored in **OneDrive**, unfortunately those files are no longer accessible after account suspension and are subsequently unable to be recovered due to encryption; even our engineers do not have standing access to the files. We know that this is not the ideal outcome in terms of your stored files, but please be assured that this is necessary for the privacy of your data and to ensure that it does not end up in the wrong hands permanently. We appreciate your understanding regarding these unfortunate circumstances.”*

As a direct result of this entire debacle, I find myself utterly infuriated with Microsoft. I simply cannot fathom how they didn't have a backup in place to restore the account. If that's truly the case, it's genuinely alarming. Because if there *are* backups, then it should have been a matter of running a few SQL scripts to restore the account. Or at least, that's how I would have designed the system in the first place. But instead, they've opted to wipe everything clean and call it a day, then head off to enjoy a good night's sleep.

This inability—or perhaps unwillingness—to resolve the issue has left me wondering... will I ever purchase anything cloud-based again? Given the way the software world is moving, I doubt there will be anything offered on physical media in the near future. But if all my subscriptions are cloud-based, then they are entirely at the mercy of the company I purchased them from. The software can be revoked at their whim, and in the event of an unfortunate incident like the hacking I experienced, access might simply vanish. And the worst part? It seems they couldn't care less if their customers lose their money, their files, or—well, pretty much anything.

It would have been at least a gesture of goodwill to say, “Yes, we're sorry about this, you need a new account, but your new account will contain all the purchases you've made, so the only thing you have to rebuild are your gaming achievements from the past few years with us.”

What's truly outrageous, at least from my perspective, is how effortlessly they dismiss the loss of "stored files." Fortunately, I didn't leave my work at the mercy of OneDrive, and though I considered it at one point, I'm now patting myself on the back for choosing a different cloud host. It's one thing to have to repurchase something, but losing access to your files—well, that could mean a lifetime's worth of memories, photos of your baby's first steps, or a book you've been writing for the last year, or even your financial records saved for the tax authorities. And to hear, “Unfortunately, those files are no longer accessible,” is truly disappointing.

What could I have been doing differently?

The person that I am most furious with in the entire process is certainly myself. I could have been a bit more careful (paranoid), and have some anti-virus software installed on the Windows computer of the house, which was mainly used for playing games by other members of the family. Truth is, I have trusted myself, that I gave a good education for people important to me about downloading all kind of stuff from the Internet, especially when it comes about game tutorials, cheats, and other kind of wonders. Seemingly I failed in that process.

The attack highly possibly was planned around a session stealing, where a nasty Trojan, presenting itself as a helper of gaming people, was started unknowingly to the player, and the built-in anti-virus that comes by default installed on the machine (guess it's provider) failed to identify it.

The virus just stayed in the memory, waiting for someone to log in on that machine, and to my biggest misfortune it was me who actually had some documents needed in desperate synchronization as fast as possible. From that point on it was a piece of cake from the attacker to get the session cookies, and use them at their will at a later stage.

As a direct consequence of the unfolding of the events I have heavily invested in a decent anti-virus, and what a surprise. Some downloaded "Game Helpers" contained a series of unwanted guests. Too bad that I have found out too late.



The Last Words Go to the Hacker

Yes, I'll dedicate this final paragraph to the individual who so gleefully caused all this chaos.

Dear rzfltfnh@ellamail.com, because that was the email ID you left as the recovery contact in the hacked account after wreaking all this havoc. I assume it's not your real name, as even a cat walking across a keyboard could come up with a better alias, but I need something to call you.

Grow up, mate. You might think you've done something clever by hacking into the account of someone you don't know, spreading chaos and mayhem, and feeling triumphant just because you could.



But don't. Your actions brought nothing positive into the world. Here's what you achieved:

- You destroyed the Minecraft realm of a group of friends who had spent hundreds of hours crafting their ideal village. They'd gone the extra mile to make it zombie- and creeper-proof, gathered rare resources to build their dream homes with unique materials. One of them had a collection of rare music discs, another had an elytra. Now, it's all gone, and they'll have to start from scratch.

- You erased our achievements in *Sea of Thieves*, painstakingly accumulated over the past two and a half years. Several rare items, now impossible to find, have vanished, and all the hard work we put in was for nothing.

Those are just the bigger issues. I won't even bother mentioning financial loss or the

the countless hours wasted dealing with customer service—it's water under the bridge now.

And yet, I bear you no ill will, not even wishing that a seagull with diarrhea empties its stomach on your head on a sunny day, because I simply feel sorry for you. All the knowledge and dedication you've gathered over the years, only to use it to destroy what others have built. Why not channel that energy into something positive and actually make a difference in the world?

Have a sunny day, my friend.

Subject: Fancy Yourself a Retro Computing Buff? Come Write for Us!

Dear Potential Contributor (or Glorious Nostalgia Nerd),

Do you harbour an unhealthy obsession with 8-bit bleeps, flickering CRTs, or the halcyon days when programming meant actual typing rather than dragging blocks around a screen? Do tales of tape drives and floppy disks bring a tear to your eye (and not just from the dust)? Splendid – you might just be one of us.

We're putting together a charming little digital magazine dedicated to retro computing, classic gaming, and all things gloriously outdated. And guess what? We want you to contribute. Not for fame. Not for fortune. (Let's be honest – we haven't even got a coffee budget.) No, this is purely for the love of the game, the joy of tinkering, and the sweet satisfaction of saying "Back in my day..."

We're especially keen on:

- Deep dives into classic games and consoles
- The dark art of programming these ancient beasts
- Hidden gems and tragically overlooked masterpieces
- Bragging rights: show off your lovingly hoarded hardware
- Misty-eyed gaming anecdotes and borderline therapy sessions
- Tales of restoration, preservation, and epic debugging battles

In return, you'll gain bragging rights, eternal respect from fellow pixel pushers, and a front-row seat in the community of retro die-hards who refuse to let good silicon die quietly.

If you fancy joining our merry band, drop us a line with your article idea, a writing sample (or an entire magnum opus, if you've already got one burning a hole in your hard drive), and we'll take it from there. It's all digital, so no trees will be harmed, and no layout editor will weep over margins.

Let's keep the magic alive – one clunky joystick and one clashing palette at a time.


Cheers, The Editorial Team.

Reachable at: retrowtf25@gmail.com

WRITERS



RETROWTF NEEDS YOU!



In our next issue you shall be
delighted by:

- Retro Game Review: The Incredible Machine
- Quake 1 through time
- Flavours of DosBox
- The Almighty Battery: Destroyer of Boards
- The card that started all: The clicking Orchid

... and some more, if all goes
according to plan

9%n\$βμ»α#
A†3#œπ

(Next issue: When it's done)