# Writing Toy Software Is A Joy

Originalartikel

Backup

<html> <div class=„row box“><p>Why you should write more toy programs</p><small>2025-06-15</small></div><div class=„row box“><p>I am a huge fan of Richard Feyman&#8217;s famous quote:</p><blockquote><p>&#8220;What I cannot create, I do not understand&#8221;</p></blockquote><p>I think it&#8217;s brilliant, and it remains true across many fields (if you&#8217;re willing to be a little creative with the definition of &#8216;create&#8217;). It is to this principle that I believe I owe everything I&#8217;m truly good at. Some will tell you to avoid reinventing the wheel, but they&#8217;re wrong: you <em>should</em> build your own wheel, because it&#8217;ll teach you more about how they work than reading a thousand books on them ever will.</p><p>In 2025, the beauty and craft of writing software is being eroded. AI is threatening to replace us (or, at least, the most joyful aspects of our craft) and software development is being increasingly commodified, measured, packaged, and industrialised. Software development needs more simple joy, and I&#8217;ve found that creating toy programs is a great way to remember why I started working with computers again.</p><h2>Keep it simple</h2><p>Toy programs follow the 80:20 rule: 20% of the work, 80% of the functionality. The point is <em>not</em> to build production-worthy software (although it is true that some of the best production software began life as a toy). Aggressively avoid over-engineering, restrict yourself to only whatever code is necessary to achieve your goal. Have every code path panic/crash until you&#8217;re forced to implement it to make progress. You might be surprised by just how easy it is to build toy versions of software you might previously have considered to be insummountably difficult to create.</p><h2>Other benefits</h2><p>I&#8217;ve been consistently surprised by just how often some arcane nugget of knowledge I&#8217;ve acquired when working on a toy project has turned out to be immensely valuable in my day job, either by giving me a head-start on tracking down a problem in a tool or library, or by recognising mistakes before they&#8217;re made.</p><p>Understanding the constraints that define the shape of software is vital for working with it, and there&#8217;s no better way to gain insight into those constraints than by running into them head-first. You might even come up with some novel solutions!</p><h2>The list</h2><p>Here is a list of toy programs I&#8217;ve attempted over the past 15 years, rated by difficulty and time required. These ratings are estimates and assume that you&#8217;re already comfortable with at least one general-purpose programming language and that, like me, you tend to only have an hour or two per day free to write code. Also included are some suggested resources that I found useful.</p><h3>Regex engine (difficulty = 4/10, time = 5 days)</h3><p>A regex engine that can read a POSIX-style regex program and recognise strings that match it. Regex is simple yet shockingly expressive, and writing a competent regex engine will teach you everything you need to know about using the language too.</p><ul><li><a href=„https://en.wikipedia.org/wiki/Regular_expression#Syntax“>Wikipedia: Regex</a></li></ul><h3>x86 OS kernel (difficulty = 7/10, time = 2 months)</h3><p>A multiboot-compatible OS kernel with a simple CLI, keyboard/mouse driver, ANSI escape sequence support, memory manager, scheduler, etc. Additional challenges include writing an in-memory filesystem, user mode and process isolation, loading ELF executables, and supporting enough video hardware to render a GUI.</p><ul><li><a href=„https://wiki.osdev.org/“>OS Dev Wiki</a></li></ul><p><a href=„https://gitlab.com/zesterer/tupai“><img src=„https://gitlab.com/zesterer/tupai/-/raw/master/doc/images/tupai-0-5-0.png“ alt=„Tupai“ referrerpolicy=„no-referrer“ /></a></p><h3>GameBoy/NES emulator (difficulty = 6/10, time = 3 weeks)</h3><p>A crude emulator for the simplest GameBoy or NES games. The GB and the NES are

classics, and both have relatively simple instruction sets and peripheral hardware. Additional challenges include writing competent PPU (video) and PSG (audio) implementations, along with dealing with some of the more exotic cartridge formats.</p><ul><li><a href=„https://gbdev.io“>GB Dev</a></li><li><a href=„https://www.nesdev.org/wiki/Nesdev_Wiki“>NES Dev Wiki</a></li></ul><h3>GameBoy Advance game (difficulty = 3/10, time = 2 weeks)</h3><p>A sprite-based game (top-down or side-on platform). The GBA is a beautiful little console to write code for and there&#8217;s an active and dedicated development community for the console. I truly believe that the GBA is one of the last game consoles that can be fully and completely understood by a single developer, right down to instruction timings.</p><ul><li><a href=„https://www.coranac.com/tonc/text/toc.htm“>Tonc</a></li><li><a href=„https://problemkaputt.de/gbatek.htm“>GBATEK</a></li></ul><h3>Physics engine (difficulty = 5/10, time = 1 week)</h3><p>A 2D rigid body physics engine that implements Newtonian physics with support for rectangles, circles, etc. On the simplest end, just spheres that push away from one-another is quite simple to implement. Things start to get complex when you introduce more complex shapes, angular momentum, and the like. Additional challenges include making collision resolution fast and scaleable, having complex interactions move toward a steady state over time, soft-body interactions, etc.</p><h3>Dynamic interpreter (difficulty = 4/10, time = 1-2 weeks)</h3><p>A tree-walking interpreter for a JavaScript-like language with basic flow control. There&#8217;s an unbounded list of extra things to add to this one, but being able to write programs in my own language still gives me child-like elation. It feels like a sort of techno-genesis: once you&#8217;ve got your own language, you can start building the universe within it.</p><ul><li><a href=„https://craftinginterpreters.com/“>Crafting Interpreters</a></li></ul><p><a href=„https://github.com/zesterer/forge“><img src=„https://blog.jsbarretto.com/img/forge.webp“ alt=„Forge“ referrerpolicy=„no-referrer“ /></a></p><h3>Compiler for a C-like (difficulty = 8/10, time = 3 months)</h3><p>A compiler for a simply-typed C-like programming language with support for at least one target archtecture. Extra challenges include implementing some of the most common optimisations (inlining, const folding, loop-invariant code motion, etc.) and designing an intermediate representation (IR) that&#8217;s general enough to support multiple backends.</p><h3>Text editor (difficulty = 5/10, time = 2-4 weeks)</h3><p>This one has a lot of variability. At the blunt end, simply reading and writing a file can be done in a few lines of Python. But building something that&#8217;s closer to a daily driver gets more complex. You could choose to implement the UI using a toolkit like QT or GTK, but I personally favour an editor that works in the console. Properly handling unicode, syntax highlighting, cursor movement, multi-buffer support, panes/windows, tabs, search/find functionality, LSP support, etc. can all add between a week or a month to the project. But if you persist, you might join the elite company of those developers who use an editor of their own creation.</p><p><a href=„https://github.com/zesterer/zte“><img src=„https://github.com/zesterer/zte/raw/master/misc/screenshot.png“ alt=„ZTE“ referrerpolicy=„no-referrer“ /></a></p><h3>Async runtime (difficulty = 6/10, time = 1 week)</h3><p>There&#8217;s a lot of language-specific variability as to what &#8216;async&#8217; actually means. In Rust, at least, this means a library that can ingest

```
impl Future
```

tasks and poll them concurrently until completion. Adding support for I/O waking makes for a fun challenge.</p><h3>Hash map (difficulty = 4/10, time = 3-5 days)</h3><p>Hash maps (or sets/dictionaries, as a higher-level language might call them) are a programmer&#8217;s bread &amp; butter. And yet, surprisingly few of us understand how they really work under the bonnet. There are a plethora of techniques to throw into the mix too: closed or open addressing, tombstones, the robin hood rule, etc. You&#8217;ll gain an appreciation for when and why they&#8217;re fast, and also when you should just use a vector + linear search.</p><ul><li><a

href=„https://www.sebastiansylvan.com/post/robin-hood-hashing-should-be-your-default-hash-table-implementation/“>Robin Hood Hashing should be your default Hash Table implementation</a></li></ul><h3>Rasteriser / texture-mapper (difficulty = 6/10, time = 2 weeks)</h3><p>Most of us have played with simple 3D graphics at some point, but how many of us truly understand how the graphics pipeline works and, more to the point, how to fix it when it doesn&#8217;t work? Writing your own software rasteriser will give you that knowledge, along with a new-found appreciation for the beauty of vector maths and half-spaces that have applications across many other fields. Additional complexity involves properly implementing clipping, a Z-buffer, N-gon rasterisation, perspective-correct texture-mapping, Phong or Gouraud shading, shadow-mapping, etc.</p><ul><li><a href=„https://www.scratchapixel.com/“>Scratch-A-Pixel</a></li><li><a href=„https://github.com/ssloy/tinyrenderer/wiki/Lesson-0:-getting-started“>How OpenGL works: software rendering in 500 lines of code</a></li></ul><p><a href=„https://github.com/zesterer/euc“><img src=„https://github.com/zesterer/euc/raw/master/misc/example.png“ alt=„euc“ referrerpolicy=„no-referrer“ /></a></p><h3>SDF Rendering (difficulty = 5/10, time = 3 days)</h3><p>Signed Distance Fields are a beautifully simple way to render 3D spaces defined through mathematics, and are perfectly suited to demoscene shaders. With relatively little work you can build yourself a cute little visualisation or some moving shapes like the graphics demos of the 80s. You&#8217;ll also gain an appreciation for shader languages and vector maths.</p><ul><li><a href=„https://iquilezles.org/articles/distfunctions/“>Inigo Quilez&#8217;s Site</a></li><li><a href=„https://www.shadertoy.com/“>ShaderToy</a></li></ul><p><a href=„https://www.shadertoy.com/view/ftXBWs“><img src=„https://blog.jsbarretto.com/img/sdf-shapes.webp“ alt=„Signed Distance Fields“ referrerpolicy=„no-referrer“ /></a></p><h3>Voxel engine (difficulty = 5/10, time = 2 weeks)</h3><p>I doubt there are many reading this that haven&#8217;t played Minecraft. It&#8217;s surprisingly easy to build your own toy voxel engine cut from a similar cloth, especially if you&#8217;ve got some knowledge of 3D graphics or game development already. The simplicity of a voxel engine, combined with the near-limitless creativity that can be expressed with them, never ceases to fill me with joy. Additional complexity can be added by tackling textures, more complex procedural generation, floodfill lighting, collisions, dynamic fluids, sending voxel data over the network, etc.</p><ul><li><a href=„https://0fps.net/2012/06/30/meshing-in-a-minecraft-game/“>0 FPS: Meshing in a Minecraft Game</a></li></ul><h3>Threaded Virtual Machine (difficulty = 6/10, time = 1 week)</h3><p>Writing interpreters is great fun. What&#8217;s more fun? <em>Faster interpreters</em>. If you keep pushing interpreters as far as they can go without doing architecture-specific codegen (like AOT or JIT), you&#8217;ll eventually wind up (re)discovering <em>threaded code</em> (not to be confused with multi-threading, which is a very different beast). It&#8217;s a beautiful way of weaving programs together out highly-optimised miniature programs, and a decent implementation can even give an AOT compiler a run for its money in the performance department.</p><ul><li><a href=„https://en.wikipedia.org/wiki/Threaded_code“>Wikipedia: Threaded code</a></li><li><a href=„https://muforth.dev/threaded-code/“>muforth.dev: Threaded code</a></li></ul><h3>Your Own GUI Toolkit (difficulty = 6/10, time = 2-3 weeks)</h3><p>Most of us have probably cobbled together a GUI program using tkinter, GTK, QT, or WinForms. But why not try writing your GUI toolkit? Additional complexity involves implementing a competent layout engine, good text shaping (inc. unicode support), accessibility support, and more. Fair warning: do not encourage people to use your tool unless it&#8217;s <em>battle-tested</em> - the world has enough GUIs with little-to-no accessibility or localisation support.</p><p><a href=„https://github.com/zesterer/gui“><img src=„https://github.com/zesterer/gui/raw/master/misc/example.png“ alt=„GUI“ referrerpolicy=„no-referrer“ /></a></p><h3>Orbital Mechanics Sim (difficulty = 6/10, time = 1 week)</h3><p>A simple simulation of Newtonian gravity can be cobbled together in a fairly short time. Infamously, gravitational systems with more than two bodies cannot be solved analytically, so you&#8217;ll have to get familiar with iterative <em>integration</em> methods. Additional complexity comes with implementing more precise and faster integration methods, accounting for relativistic effects, and writing a visualiser. If you&#8217;ve got the maths right, you can even try plugging in real numbers from NASA to predict the next high tide or full moon.</p><ul><li><a href=„https://en.wikipedia.org/wiki/Leapfrog_integration“>Wikipedia: Leapfrog integration</a></li></ul><h3>Bitwise Challenge (difficulty = 3/10, time = 2-3 days)</h3><p>Here&#8217;s one I came up with for myself, but I think it would make for a great game jam: write a game that only persists 64 bits of state between subsequent frames. That&#8217;s 64 bits for everything: the entire frame-for-frame game state should be reproducible using only 64 bits of data. It sounds simple, but it forces you to get incredibly creative with your game state management. Details about the rules can be found on the GitHub page below.</p><ul><li><a href=„https://github.com/zesterer/the-bitwise-challenge“>The Bitwise Challenge</a></li></ul><p><a href=„https://github.com/zesterer/bitwise-examples“><img src=„https://blog.jsbarretto.com/img/snake.webp“ alt=„Snake“ referrerpolicy=„no-referrer“ /></a></p><h3>An ECS Framework (difficulty = 4/10, time = 1-2 weeks)</h3><p>For all those game devs out there: try building your own <a href=„https://en.wikipedia.org/wiki/Entity_component_system“>ECS</a> framework. It&#8217;s not as hard as you might think (you might have accidentally done it already!). Extra points if you can build in safety and correctness features, as well as good integration with your programming language of choice&#8217;s type system features.</p><p>I built a custom ECS for my <a href=„https://www.youtube.com/watch?v=nS5rj80L-pk“>Super Mario 64 on the GBA</a> project due to the unique performance and memory constraints of the platform, and enjoyed it a lot.</p><p>youtube_ns5rj80l-pk_gt</p><h3>CHIP-8 Emulator (difficulty = 3/10, time = 3-6 days)</h3><p>The <a href=„https://en.wikipedia.org/wiki/CHIP-8“>CHIP-8</a> is a beautifully simple virtual machine from the 70s. You can write a fully compliant emulator in a day or two, and there are an enormous plethora of fan-made games that run on it. <a href=„https://github.com/zesterer/emul8/raw/refs/heads/master/test/test.ch8“>Here&#8217;s</a> a game I made for it.</p><ul><li><a href=„https://en.wikipedia.org/wiki/CHIP-8“>Wikipedia: CHIP-8</a></li></ul><p><a href=„https://github.com/zesterer/emul8“><img src=„https://github.com/zesterer/emul8/raw/master/misc/screenshot.png“ alt=„Emul8“ referrerpolicy=„no-referrer“ /></a></p><h3>Chess engine (difficulty = 5/10, time = 2-5 days)</h3><p>Writing a chess engine is great fun. You&#8217;ll start off with every move it makes being illegal, but over time it&#8217;ll get smart and smarter. Experiencing a loss to your own chess engine really is a rite of passage, and it feels magical.</p><ul><li><a href=„https://en.wikipedia.org/wiki/Minimax“>Wikipedia: Minmax</a></li><li><a href=„https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning“>Wikipedia: Alpha-beta pruning</a></li></ul><p><img src=„https://blog.jsbarretto.com/img/chess.webp“ alt=„Chess“ referrerpolicy=„no-referrer“ /></p><h3>POSIX shell (difficulty = 4/10, time = 3-5 days)</h3><p>We interact with shells every day, and building one will teach you can incredible amount about POSIX - how it works, and how it doesn&#8217;t. A simple one can be built in a day, but compliance with an existing shell language will take time and teach you more than you ever wanted to know about its quirks.</p><ul><li><a href=„https://brennan.io/2015/01/16/write-a-shell-in-c/“>Write a shell in C</a></li></ul><p><a href=„https://github.com/zesterer/tosh“><img src=„https://raw.githubusercontent.com/zesterer/tosh/master/misc/screen0.png“ alt=„Tosh“ referrerpolicy=„no-referrer“ /></a></p><h2>A note on learning and LLMs</h2><p>Perhaps you&#8217;re a user of LLMs. I get it, they&#8217;re neat tools. They&#8217;re useful for certain kinds of learning. But I might suggest resisting the temptation to use them for projects like this. Knowledge is not supposed to be fed to you on a plate. If you want that sort of learning, read a book - the joy in building toy projects like this comes from an exploration of the unknown, without polluting one&#8217;s mind with an existing solution. If you&#8217;ve been using LLMs for a while, this cold-turkey approach might even be painful at first, but persist. There is no joy without pain.</p><p>The runner&#8217;s high doesn&#8217;t come to those that take the bus.</p></div> </html>