

HACKERMONTHLY

Issue 38 July 2013

The Apprentice Programmer

Tobias Lütke



```
{  
  join: 'Intensive Online Bootcamp',  
  learn: 'Web Development',  
  goto: 'http://www.gotealeaf.com'  
}
```



Tealeaf Academy
an online school for developers

Learn Ruby on Rails | Level up Skills | Launch Products | Get a Job



Now you can hack on DuckDuckGo

DuckDuckHack

Create instant answer plugins for DuckDuckGo

duckduckhack.com

Curator

Lim Cheng Soon

Contributors

Tobias Lütke
George W. Hart
Basalgangster
Patrick Wyatt
Evan Travers
Geoffroy Couprie
Mike Bostock
Mario Livio
Glenn Reid

Proofreaders

Emily Griffin
Sigmarie Soto

Ebook Conversion

Ashish Kumar Jha

Printer

MagCloud

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com*, a social news website wildly popular among programmers and startup founders. The submission guidelines state that content can be “anything that gratifies one’s intellectual curiosity.” Every month, we select from the top voted articles on Hacker News and print them in magazine format. For more, visit *hackermonthly.com*

Advertising

ads@hackermonthly.com

Contact

contact@hackermonthly.com

Published by

Netizens Media
46, Taylor Road,
11600 Penang,
Malaysia.



Cover: Tobias Lütke

Hacker Monthly is published by Netizens Media and not affiliated with Y Combinator in any way.

Contents

FEATURES

06 The Apprentice Programmer

By TOBIAS LÜTKE

09 How to Slice a Bagel into Two Linked Halves

By GEORGE W. HART



PROGRAMMING

12 Building Photoshop

By BASALGANGSTER

20 The StarCraft Path-finding Hack

By PATRICK WYATT

24 Workflow in Tmux

By EVAN TRAVERS

26 Tips to Accelerate SSL

By GEOFFROY COUPRIE

31 Why Use Make

By MIKE BOSTOCK

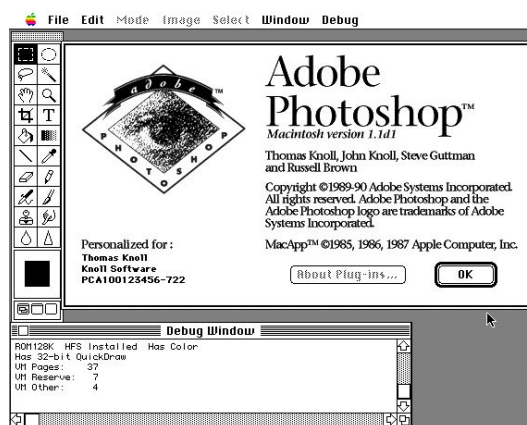
SPECIAL

34 Where and When Did the Symbols “+” and “-” Originate?

By MARIO LIVIO

36 What It’s Really Like Working With Steve Jobs

By GLENN REID





The Apprentice Programmer

By TOBIAS LÜTKE

I DROPPED OUT OF school when I was 16 years old. School was not for me. To me, computers were so much more interesting. Right or wrong, I felt like I wasted my time there and my real education was starting when I came home. I lost respect for the institution and of course this meant that I no longer bothered to put any effort into it. They diagnosed me with all sorts of learning disabilities and started to medicate me. I wanted to leave it all behind.

I decided the best thing to do was to drop out and start an apprenticeship as a Fachinformatiker — computer programmer. This might sound like a stupid decision to people in North America, who often go to College or University to get a degree in something like computer science, but in Germany, leaving high school for an apprenticeship is not out of the ordinary. It

is called the dual education system, and it is likely one of the main reasons for Germany's success.

The system has its roots in the history of the region. Carpenters and a number of other important crafts have used an apprenticeship system to teach and build expertise for hundreds, if not thousands of years. The underlying idea is that there are professions that profit more from experience than theoretical understanding, and that education time is far better spent doing the actual work than by watching or listening.

Many companies in Germany take on apprentices, much like North American companies accept interns and co-op students. If a company decides to take you on as an apprentice, the position is guaranteed by the state. Should the company go bust, you are placed with another company the next

day. There is a web of companies guaranteeing the positions for each other, spread all across the country.

Unlike interns in North American companies, apprentices in Germany are treated like normal junior employees except they are cheaper, (700 marks per month or \$400USD when I went). They're also often younger and miss about 60 work-days a year to attend classes at their vocational school. The schools teach the theory behind each chosen trade and certify the students at the end of 3 years with an exit exam which concludes the program. Student apprentices, (they called us "Stift" = Pen), who successfully complete the program and pass the exam earn the vocational title.

“Most days I came to work and found a printout of the code I wrote the day before annotated with red marker everywhere.”

I landed an apprenticeship with a company in my hometown called BOG Koblenz, a subsidiary of Siemens with a history of taking on apprentices. For some reason I vividly remember this one question from my interview:

The number of lilies in a pond double every day. So, on the first day of the month there is one lily. On the second day, two lilies, the next day four lilies, then eight, sixteen, thirty two, etc. If the pond is full on the 30th day of the month, what day is it half full?

That was not exactly difficult.

Three other Stifts started with me at the same time. On our first day we got a tour of the 150 person company, which seemed massive to me at the time. The first year was about paying our dues: 3 months running the cafeteria, 3 helping in accounting, 3 more working in inventory, and then 3 months at reception. It was a rite of passage, they told us.

The first 3 months in the cafeteria meant I quickly met everyone in the company and learned what kind of coffee or tea they liked. I made sure to keep them well caffeinated. My absolute favorite group

of people worked in a small room in the basement of a secondary building. I do not remember what their official titles were, but they were essentially doing Skunk Works [hn.my/skunk] down there.

They did things differently than everyone else. Most of the company used an esoteric programming environment called Rosie SQL — which seemed like death to my Demo Szene-honed sensibilities (Assembler, Pascal or bust!) — these guys used Delphi. I fell in love with it! Here was a programming language that put humans before machines. It was built for instant gratification, experimentation, and rapid prototyping. Its window management library, called the VCL, was so much greater than anything I had seen before. More importantly, it was run by Jürgen.

Jürgen was a long-haired, 50-something, grizzled rocker who would have been right at home in any Hell's Angels gang. He was a rebel. He refused to wear the company attire, refused to use the formal language [hn.my/german], and called people out on bad ideas in plain language when he saw them. Despite all of this, everyone respected him. I tried my best to

make it absolutely obvious that I wanted to work for him. I borrowed the Delphi manuals and committed them to memory in my downtime between coffee runs.

At the same time, I would attend my vocational school every Friday, and twice a year we would go for two straight weeks to study and take exams. This was a much better way for me to learn. It felt relevant. I learned the fundamentals of things picked up from being around Jürgen's team. We learned about algorithms, Big O, etc — even some basic soldering and electrical work.

It turned out those learning disabilities were not real disabilities; I was simply a kinesthetic learner. I could not understand or come up with solutions to problems I have never had. At my vocational school, I knew the problems we were solving. I had been in those situations. It was great! My self esteem and confidence improved quickly.

My plan was working.

After the first year, Jürgen drafted me to be a part of his little basement-dwelling team. It was probably the most important thing that happened to me in my professional life. Jürgen was a master teacher. He created an environment in which it was not only possible but easy to move through 10 years of career development every year. It is a method and an environment which I am fiercely trying to replicate at Shopify.

Most days I came to work and found a printout of the code I wrote the day before annotated with red marker everywhere. I used poor idioms or could have chosen better abstractions or done a better job hinting at the architecture of the overall system. This taught me not to tangle my ego up in the code I write. There are always ways to improve it and getting this feedback is a gift.

I remember we made software for GM. One particular car dealership needed a faster system to estimate the value of incoming used cars. A big competitive advantage. Jürgen gave this project to me. Shipping it meant Jürgen and I had to drive to the dealership which was a day trip away. In preparation for it, the company gave me extra money so I could buy a suit. We work for Siemens after all. We had to look the part.

The day before the installation, Jürgen casually tells me he has somewhere else to be. I would be going by myself. I felt overwhelmed but somehow managed to make a good impression and got everything working regardless.

This pattern kept on repeating itself. Jürgen somehow knew the extent of my comfort zone and manufactured situations which were slightly outside it. I overcame them through trial and error, through doing, and immediately applying the theory I was learning at the vocational school to practice at my apprenticeship, I succeeded.

My degree is not recognized in North America so I am technically a high school dropout. My cofounder at Shopify has a PhD, so we always joke that together we average out to a bachelor's degree.

Not that degrees matter anymore. They do not. Experience does. That is one of the things my apprenticeship and the dual education system in general taught me: experiencing and learning things quickly is the ultimate life skill. If you can do that, you can conjure up impossible situations for yourself over and over again and succeed.

Perhaps most importantly, the apprenticeship program gave me a solid head start. If I had gone to a University and studied to get a PhD like my cofounder, I might JUST be getting out of school. Instead, at 32 years old, I have been paid to build complex software for almost half my life.

That is a powerful concept and one within the reach of almost any German student thanks to the dual education system. At the last count, there are 356 different occupations or occupational categories which offer apprenticeships. From hair dressers to oven builders to various specializations of computer programming. For hands-on people or kinesthetic learners like me, the apprenticeship program created a legitimate path to success.

It was the perfect environment for me: I learned a lot, and I am eternally thankful to have chosen that path. If only more countries struggling with dropout rates and job creation would give their students a similar choice. ■

Tobi is CEO and founder of Shopify. Before taking the helm of the company, Tobi was active in many open source projects. The best known is the Ruby on Rails framework where Tobi served as a core team member starting in 2004. He also released a series of open source projects that are used around the world such as Liquid, DelayedJob, and ActiveMerchant.

Reprinted with permission of the original author.
First appeared in hn.my/tobi (lutke.com)

How to Slice a Bagel into Two Linked Halves

By GEORGE W. HART

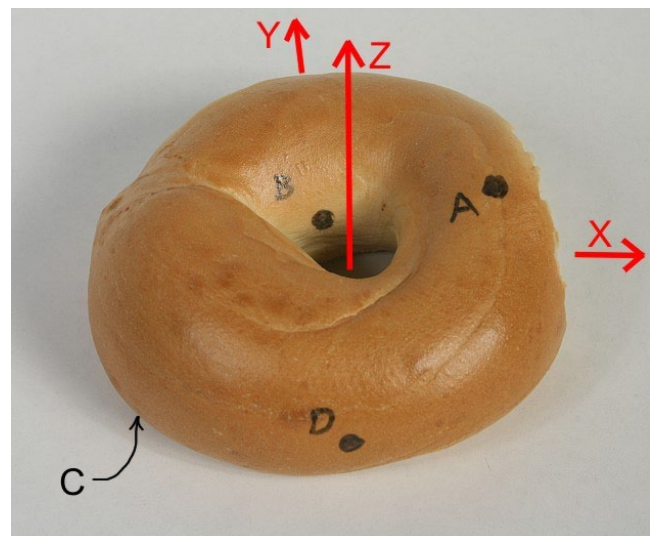


IT IS NOT hard to cut a bagel into two equal halves which are linked like two links of a chain.

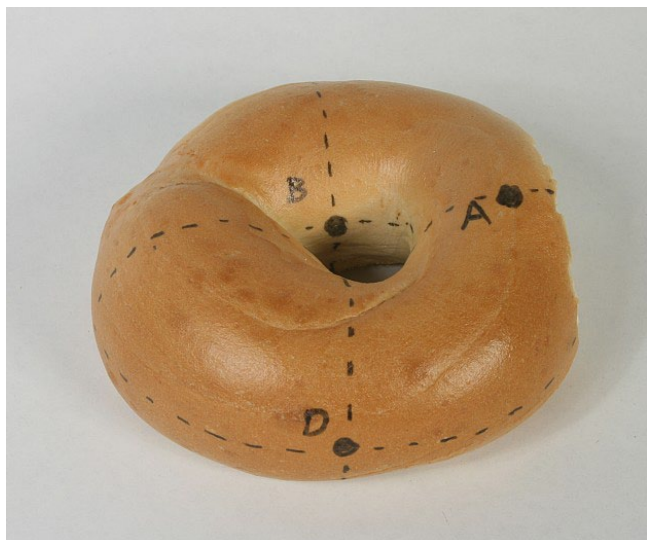
To start, you must visualize four key points. Center the bagel at the origin, circling the Z axis.

A is the highest point above the +X axis. B is where the +Y axis enters the bagel.

C is the lowest point below the -X axis. D is where the -Y axis exits the bagel.

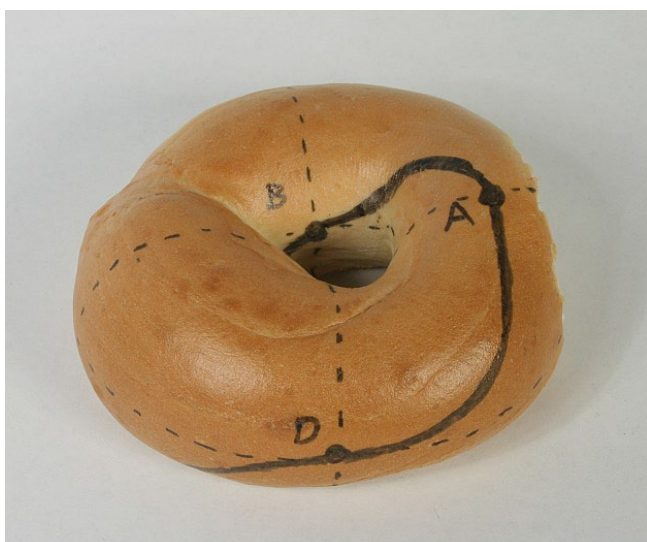


These sharpie markings on the bagel are just to help visualize the geometry and the points. You don't need to actually write on the bagel to cut it properly.



The line ABCDA, which goes smoothly through all four key points, is the cut line.

As it goes 360 degrees around the Z axis, it also goes 360 degrees around the bagel.

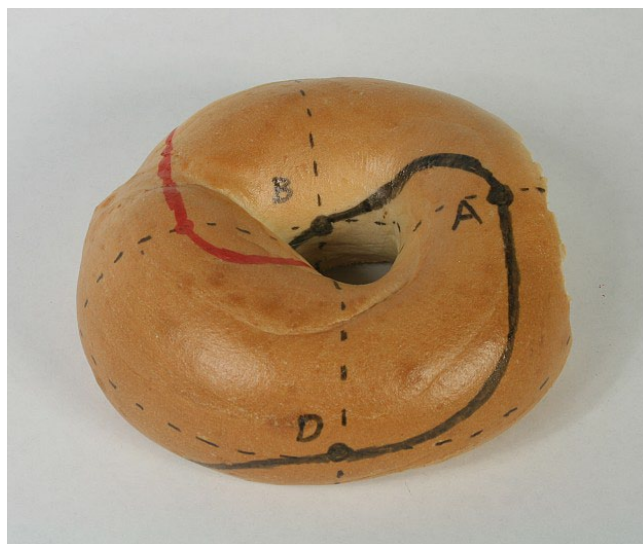


The red line is like the black line but is rotated 180 degrees (around Z or through the hole).

An ideal knife could enter on the black line and come out exactly opposite, on the red line.

But in practice, it is easier to cut halfway on both the black line and the red line.

The cutting surface is a two-twist Mobius strip; it has two sides, one for each half.



After being cut, the two halves can be moved but are still linked together, each passing through the hole of the other. (So when you buy your bagels, pick ones with the biggest holes.)



If you visualize the key points and a smooth curve connecting them, you do not need to draw on the bagel. Here the two parts are pulled slightly apart.



If your cut is neat, the two halves are congruent. They are of the same handedness. (You can make both be the opposite handedness if you follow these instructions in a mirror.)

You can toast them in a toaster oven while linked together, but move them around every minute or so. Otherwise, some parts will cook much more than others, as shown in this half.



It is much more fun to put cream cheese on these bagels than on an ordinary bagel. In addition to the intellectual stimulation, you get more cream cheese because there is slightly more surface area.



Topology problem: Modify the cut so the cutting surface is a one-twist Mobius strip. (You can still get cream cheese into the cut, but it doesn't separate into two parts.)

Calculus problem: What is the ratio of the surface area of this linked cut to the surface area of the usual planar bagel slice?

For future research: How to make Mobius lox...

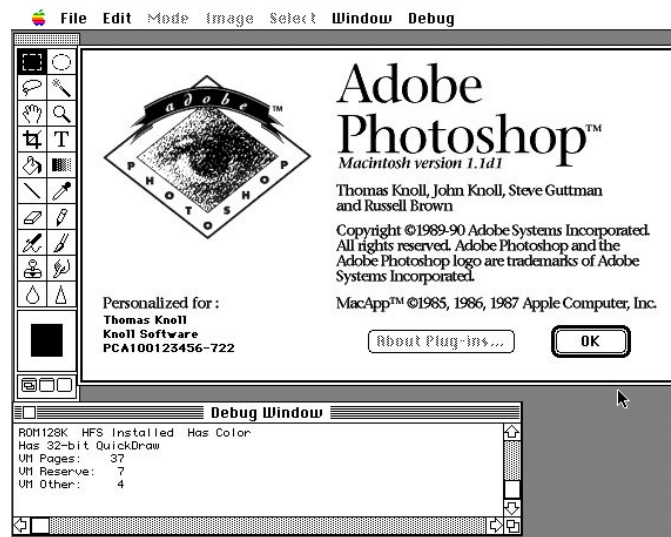
Note: I have had my students do this activity in my Computers and Sculpture class. It is very successful if the students work in pairs with two bagels per team. For the first bagel, I have them draw the indicated lines with a "sharpie." Then they can do the second bagel without the lines. (We omit the schmear of cream cheese.) After doing this, one can better appreciate the stone carving of Keizo Ushio, [hn.my/keizo] who makes analogous cuts in granite to produce monumental sculpture. ■

George Hart started hacking the internet (back when it was the Arpanet) in the early 1970s, when he worked at BBN. He has MIT degrees in mathematics, computer science, and electrical engineering and has been a professor at both Columbia University and Stony Brook University. Nowadays, he focuses on creating cool mathematical sculpture and fun videos about mathematics. See his web site, georgehart.com, for some amazing uses of 3D printing.

Reprinted with permission of the original author.
First appeared in hn.my/bagel (georgehart.com)

Building Photoshop

By BASALGANGSTER



ON FEBRUARY 13, 2013, the Computer History Museum reported that they had received the source code for version 1.1 of Photoshop from Adobe, and had permission to release it for non-commercial use. This is the second time recently that something like this has happened, the first being the release of the Quickdraw and MacPaint [hn.my/macpaint] source code by Apple last year. In both cases, there

was a nice commentary by Grady Booch, software design legend and trustee of the Computer History Museum. Both times, Booch reported having a look at the code and found it to be elegant and clear, and outstanding examples of how to write software. In his commentary on the Photoshop source code, Booch says, “Software source code is the literature of computer scientists, and it deserves to be studied and appreciated”.

Writers need to study the writing of others, and usually they do. Computer programmers? Not so much. There is a lot written about programming, and most programmers read that stuff (including Grady Booch’s books), but there are very few opportunities to actually read an acknowledged Great Work of programming from the past. So now we have the sources to Photoshop, an indisputable software masterpiece. How can we make

use of it? Shall we read through the 100,000 or so lines of code in 179 files to see how Thomas Knoll made it do its magic? Maybe Grady Booch can just look at the code and immediately understand the thread of genius that runs through it, but I can't. Source code may be the literature of computer scientists, and each program's source code may tell the story of its function, but that story is written in a scrambled, crazy order and it makes no sense. To understand their stories, computer programs must be read in the order of their execution, not the order of the text in the files. For Photoshop, as for MacPaint, that means the code has to be compiled and running when we read it. We need to see it in its natural order, and while it is executing, so we can compare what it says to what it does. Only then can we learn its story.

MacApp version 1.1

Photoshop was created for the Macintosh computer in the late 1980s, and it looks a little strange to today's programmers. Anyone can see (and many are surprised) that it is written in Pascal. Maybe it is not well known that 25 years ago the Pascal language saw a lot of use, even for teaching programming in college. Pascal is not taught much these days, and that must be why almost nobody notices that Photoshop is not written in ordinary Pascal, but rather in Object Pascal. Even fewer would recognize that it is written using Apple's MacApp object library. The version of Photoshop that was released to the public is not a complete set of sources. If you are going to build Photoshop from the distributed files, you need a Macintosh Object

Pascal compiler, its associated libraries, and the appropriate version of the MacApp class library. MacApp was not included in the source code release, and could not have been, because Adobe does not have a license to distribute MacApp™.

It would have been helpful if Apple had decided to release the source code for the appropriate version of MacApp at the same time as Photoshop. Why not? The source for MacApp is not a trade secret. The class library was distributed to programmers as source, and they were expected to compile it themselves. The version of MacApp in question is version 1, released in 1987 and replaced by a totally rewritten version in 1990. The original was distributed as a set of 2 floppy disks. MacApp version 2 was the first version to be distributed on CDROM. In 1990, Apple wanted programmers to port their programs to version 2, and they didn't include version 1 on the version 2 CD or on any of the later developer distributions. I think it is probably pretty rare. Of course, some people no doubt still have an old copy of MacApp on floppy disks. I didn't. I started using MacApp at version 2, in 1990, and never had a copy of version 1. Miraculously, a pair of MacApp v1.1.1 diskettes was for sale on ebay at the time the Photoshop code was released.



I bought them, realizing that at their age they might or might not be readable. They mostly were. One of the disks contains all of the Pascal and assembly source code for the MacApp library. The other has everything else needed to build and test the library, including the resource files, the one object file for which Apple did not share their source (called wwDriver.c.o), and the MPW script called MABuild that is used to build MacApp programs from their make files. There is also a compiled MPW tool, called PostRez, used to post-process the code. It connects menu GUI elements compiled by the resource compiler Rez to the object code that they are supposed to evoke. The second diskette also contains the classic MacApp example programs, like DrawShapes and Nothing. I am lucky; the diskettes I bought were mostly readable. The second diskette was completely error-free. The source code diskette had one unreadable file that contained seven bad sectors. The file that could not be read was UTTEView.incl.p, which contained the implementation of an object called TTEView that is a baby text editor. It was a relief to find that Photoshop did not use this component of MacApp, but I wasn't surprised. TTEView was not used much outside of Apple's demos because

it was limited to editing text blocks of size 32K or smaller.

The MacApp that I bought was complete enough to compile Photoshop. But if you happen to have a copy of UTTEView.incl.p, I'd be interested in acquiring a copy of that file, just to

“Thomas Knoll must have created Photoshop without the use of any source-level debugger other than the one that was part of MacApp.”

complete the library. Don't worry about Apple's lawyers. I have a license for MacApp.

I tried to compile MacApp and a couple of examples (the ones that didn't use TTEView) to make sure my MPW setup was working with MacApp. It wasn't. A quick look around the inter-webs yielded this post [hn.my/macapp] on *comp.sys.mac.programmer* from Apple's Keith Rollin in 1989, after the release of MacApp 2.0 (thanks to macgui.com for archiving these old usenet posts). Basically, it says that if you want to keep on using MacApp 1.x (and you shouldn't), you have to use MPW version 2.x and it's Pascal compiler, not version 3.x. Of course I was trying to use MPW 3.0. It is the earliest version I have. I haven't used MPW 2 on any computer since about 1991. And like MacApp v1, MPW 2 was only distributed on floppy diskettes. I still have the MPW 2.0 diskettes, but in a moment of bad judgment in the mid 1990s I reused them to store some files that seemed more important at the time (but don't seem so anymore). There were no MPW 2 diskettes that I could find for sale anywhere. According to the usenet post, I could use MPW 3.0 under system 6 with some changes

to the MacApp makefiles and to the application resources. It says that might suffice for builds that do not evoke MacApp's built-in debugger. But according to Keith Rollin, the MacApp debugger will not work if MacApp is compiled under MPW 3. I tried the Nothing and DrawShapes examples both ways; the nodebug versions worked and the debug versions crashed. I could build the nodebug version of Photoshop, but what good is that? I don't just want to just use some old version of Photoshop; I want to run it in the MacApp debugger. MPW version 2 didn't have a source level debugger. Apple's first source debugger for MPW (SADE) was released with version 3. I think Thomas Knoll must have created Photoshop without the use of any source-level debugger other than the one that was part of MacApp. If I want to see it the way he saw it (and I do), I would need to compile MacApp with debugging turned on. Luckily, the problem with the debugger was fixed with only a small change in the MacApp source code.

Getting MacApp to Build

If you build a MacApp v1.1 program with debug on, it enables code for the Writeln window, names in code, range checking, the MacApp debugger interface, and subroutine tracing. Tracing subroutine calls, when on, means that the MacApp debugger keeps track of entry into and exit from every subroutine. When trace is then activated in the debugger, every subroutine call is recorded to the Writeln window, similarly to the trace function in Lisp. If the code was compiled for the debugger, whether tracing is turned on or not, MacApp calls a couple of weirdly named functions called `%_BP` and `%_EP` respectively (implemented in unit `UTrace.incl.p`) on entry and exit to every subroutine. Some subroutines that should never be seen in a trace (like the ones that write trace information to the Writeln window) are exempted from `%_BP` and `%_EP` using the directive `{SD+}`, which leaves subroutine names in code but does not trace. The directive `{SD++}` turns tracing back on for the next subroutine. In MacApp v1.1, both `%_BP` and `%_EP` call a subroutine, named `MeasureTally`. For some reason, `MeasureTally` was not exempt from the trace. I don't

“The files in the Photoshop distribution are not ready to use. All the text files are contaminated by Windows-style newline characters.”

know what would happen if it was compiled with the MPW 2.0 Pascal compiler, but MPW 3.0 compiled this into an infinite recursion, as it should. At startup any program compiled with debugging on would immediately hang, stay hung for a while, and then the stack would collide with something vital in the heap (probably some code segment) and there would be a Bad Crash. It was an easy bug to track down. As soon as it hung but before it crashed, I hit the programmer's switch and fell into TMON. I was in %_EP. Both %_EP and MeasureTally are short, and I could step through them and see them call each other in turn. The problem was fixed by exempting MeasureTally from the trace.

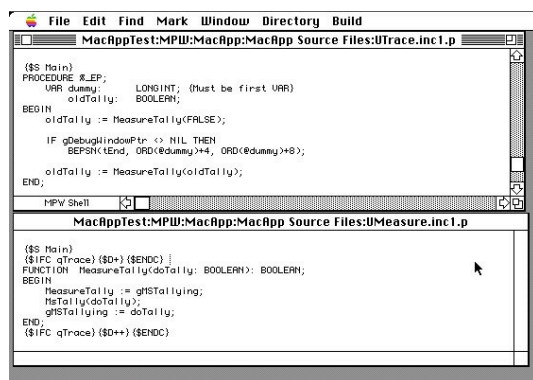
I don't see how the MacApp debugger could have ever worked without this change. It seems to me that everything in UMeasure probably should have been exempted, but it doesn't look like it was. Anyway, just exempting MeasureTally got everything working for now. In pretty short order, I was able to build MacApp in System 6 using MPW 3.0 and get the examples to work with debugging turned on. It was fun messing around with the MacApp example programs. What a thrill. I was ready to compile Photoshop.

Fixing the Files

The files in the Photoshop distribution are not ready to use. Firstly, all the text files are contaminated by

the left limit, and then advance the paper one line. This made sense as a line terminator back in the days of the PDP-11, but by the time the Macintosh was released it was already vestigial. Macintoshes were never expected to be connected to teletype machines. They needed a character to represent newline, but one was enough. So Apple omitted the linefeed and kept the carriage return. Somebody at Adobe or at the Computer Museum must have thought we would be using this code with Windows, or some other operating system intended to work with teletype machines. To build Photoshop, the first thing you have to do is to strip out all those extraneous linefeeds. Next, it is necessary to attend to the copyright blurb at the top of each of the resource files. Every Pascal and every Rez file has a little snippet of legalese at the top, encased in curly brackets, like this:

*{Photoshop version 1.0.1, file:
About.r Computer History
Museum, www.computerhistory.
org This material is (C) Copyright
1990 Adobe Systems Inc. It may
not be distributed to third parties. It
is licensed for non-commercial use
according to www.computerhistory.
org/softwarelicense/photoshop/ }*



Windows-style newline characters. Every text line in Windows (and in DOS before it, and in CPM and DEC RT-11 before that) was terminated by a pair of characters, a carriage return (ascii 13) and a linefeed (ascii 10). If you were printing the text using a teletype machine these two characters would first move the print head to

“The resource files are in the Rez language, whose conventions are apparently unknown to the archivers, because they retained the curly bracket Pascal comments.”

Curly brackets delimit comments in Pascal, so these are fine for the Pascal files. For the assembly language files, the curly brackets have already been replaced with semicolons at the start of each line, which is exactly right. Semicolons start a comment line in MPW assembly (as in many others). But the resource files are in the Rez language, whose conventions are apparently unknown to the archivers, because they retained the curly bracket Pascal comments. Rez resembles C, so curly brackets mean something altogether different. Comment lines in Rez begin with `//`, and so all of these need to be added. Finally, the MPW make files and other shell scripts used a unique lexicon of special characters. Comment lines began with `#`, as they do in most unix shell scripts, and this part has been done right in the sources as distributed. But all the MPW scripts are mangled by the use of a backslash (`\`) instead of option-d (`ð`) as the escape character (for line continuation), and all instances of option-f (`f`) to indicate target-dependency in makefiles have been replaced with colon (`:`), I suppose because that is what is used in unix makefiles. Fixing all that was easily done with a couple of little MPW scripts.

There are some additional small version issues to address before compiling Photoshop in MPW 3.0. The Photoshop code uses a Pascal interface file called `Quickdraw32bit.p`. The contents of this file were folded into `Quickdraw.p` at the time of release of MPW 3, so you have to remove the reference to `Quickdraw32bit.p` in several places. The MPW 3.0 version of `Quickdraw.p` left out the definition of a couple of constants that were in `Quickdraw32bit.p`, however, so I had to borrow a later version of `Quickdraw.p` from MPW 3.2. Also, I think Knoll must have been using a particularly early version of the `Quickdraw32bit.p` file. He refers to a field of the `ColorTable` record as `transindex`. That field is called `ctFlags` in *Inside Macintosh V*, and in the MPW 3.x interface files. I changed references to `transindex` with `ctFlags` everywhere it appeared in the Photoshop code.

Knoll's Modifications of MacApp

One of the great things about MacApp was that it was distributed as source code. The first thing you had to do was to build the library yourself. What was so great about that? Well, it meant you could improve it if you wanted to. In this particular case that was not so great. I do not have a copy of Thomas Knoll's version of MacApp, so I was hoping that he had used an unaltered version, but I knew it wasn't likely. A lot of things you can readily see happening in Photoshop were not implemented in MacApp v1. For example, that floating palette window with the tools in it. Floating windows like that were (in)famously not part of MacApp v1. Another well-known difficulty with MacApp at that time was its lack of support for "mouse-up tracking." This problem was even mentioned by Kurt Schmucker in his 1986 book on MacApp, "Object-Oriented Programming for the Macintosh."

To understand the problem with mouse-up tracking, you have to understand MacApp's support for mouse tracking. Mouse tracking means that your code is drawing and redrawing something on the screen in response to movements of

“When your user was drawing, mouse monitoring code would call your command object and tell it where the mouse was and where it had been most recently.”

the mouse. Normally this happens only when the mouse button is down. When your user was drawing, or defining a selection or dragging something, or for whatever reason moving the mouse around in a window with the button pressed, mouse monitoring code (in the TFrame object) would call your command object (subclass of TCommand) and tell it where the mouse was and where it had been most recently. A command object that handled mouse tracking was called a tracker. You were expected to subclass TCommand to implement each specific behavior of your program during mouse-down tracking, but you didn't expect to have to change TFrame at all. TFrame would call your tracker periodically while the user held the mouse button down, so you could update what was needed and draw stuff on the screen accordingly. When the user let go of the mouse button, your tracker was called one last time, with the advisement this was the end of the series. At that time you were supposed to clean up and draw the final state of your window as it should appear after the user-interaction.

It was a cool system, because you just made a command object and didn't worry about how it got called. Stuff that happened in between the time that the TApplication object intercepted the user's action and the time your tracker was called stayed completely invisible to you as a programmer. The down side was you couldn't easily keep tracking the mouse after a mouse-up event. The TFrame would think that the action was over and wouldn't call your tracker again till the mouse button was pressed to start a new interaction. But let's say you wanted to let the user draw with the mouse up. For example, imagine you wanted the user to be able to define a polygon by clicking on a series of vertices, and have the program drag out a dynamic line between your last clicked point and the current mouse position as the user moved around with the mouse button up. Mouse-up tracking was used in the polygon-drawing tools in MacDraw, SuperPaint, and Canvas, and lots of other drawing programs. If you wanted to do mouse-up drawing in a MacApp v1 application, you could not do it purely in your command object. You would have to bore down deep and alter the

normally untouched MacApp code in class TFrame that called command objects. These days this kind of problem is solved using multiple inheritance, or (even better) by delegation, so you never have to alter the guts of the framework, and this is why Apple doesn't have to share the class framework source code with you.

It turned out that the floating palette window was not a problem. Knoll coded that window's behavior by subclassing the existing MacApp TView class. That's one of the things I'd like to study when I get everything running. And Photoshop almost did not use mouse-up tracking. But it did, just for one small thing. I am a long-time Photoshop user, but I didn't know or had forgotten about this feature. I learned about it because I got an error at compile-time. A subclass of TCommand (a tracker) in Photoshop, called TLassoSelector, overrides a TCommand method called TrackMouseUp. The compiler knew that in my version of MacApp, TCommand does not have a message called TrackMouseUp. In Thomas Knoll's version, there must have been.

The Lasso tool uses mouse-up tracking, but only sometimes. Try this in any version of Photoshop. Start to make a selection using the Lasso tool. Part way through the selection, press the option key, and then release the mouse button. Photoshop drags out a straight line between the place you released the mouse button and the current location of the mouse. If you release the option key, it closes the selection with a straight line. Photoshop v1.1 also did that. Knoll had to change MacApp to make this happen, and I didn't have a copy of his changes. Of course it is easy to put a dummy method called `TrackMouseUp` in `TCommand`; that would satisfy the compiler. But it would never get called, and the mouse-up tracking would not happen. My version of Photoshop would compile, but it would not be authentic. Not good enough. I had to add a call to `TrackMouseUp` somewhere in just the right spot. I would know it was the right spot, because if I did it right, `TrackMouseUp` in Knoll's lasso command object would do the right thing. Of course, I also needed a default do-nothing version of `TrackMouseUp` in `TCommand` so it could be overridden. I'm pretty sure `TFrame` is the object that has to call `TrackMouseUp`. In MacApp v1 (but not in later versions), objects of class `TFrame` provide scrolling to views in windows. They also monitor clicks and drags in their screen territories, and send messages to tracker command objects that are supposed to be drawing in those spaces. As pointed out by Grady Booch in his commentary, there are almost no comments at all in the Photoshop code; there was no hint left anywhere by Knoll. By the looks of it, `TrackMouseUp` in

the `LassoSelector` was supposed to take over for `TFrame` after `TFrame` gave up tracking the mouse. I stuck a call to `TrackMouseUp` in the part of `TFrame.TrackInContent` that runs at the end of tracking, when the mouse has just been released. It works, but I can't be sure whether this is the same change that was made to MacApp for the original Photoshop. A similar change to another part `TFrame` was also required because of another mystery override. Photoshop overrides `TFrame.CalcSBarMin`, another MacApp method that does not exist. There is a `TFrame.CalcSBarMax` in `TFrame.AdjustSBars`, and so I put the call to `CalcSBarMin` there too. I don't actually know what the function of this is, so I can't even be sure it's working. I'll find that later when I study it in action. Photoshop also expects to find a procedure called `FlashButton`. I thought it was apparent what this should do and wrote something accordingly.

EVEITF.o

Photoshop expects to link an object file called `EVEITF.o`. The file is not there. It apparently provides the code for 5 functions, called `EVEStatus`, `EVEReset`, `EVEEnable`, `EVEReadGPR`, and `EVEChallenge`. These functions are called by a global function, `VerifyEve`, which is present, and is called at startup time. Linking Photoshop fails because of these unresolved references. Examination of `VerifyEve` shows that none of the EVE functions are called unless Photoshop has a resource of type "Eve" (that's E-v-e-space). There is no such resource in the distribution's resource description file, so these functions would never be called anyway. The source

code distribution is designed to link `EVEITF.o` into the final build, but not to call it. It is dead code. But adding an appropriate "Eve" resource would bring it back to life without a rebuild. I think `VerifyEve` must have had something to do with serial number registration, because it is called immediately after `RegisterCopy`, which checks whether or not a valid registration number has been entered. After commenting out the EVE functions, the program runs and `RegisterCopy` works fine. It successfully accepts valid Photoshop serial numbers and rejects invalid ones, even without `VerifyEve`. So `VerifyEve` remains a mystery. The personalization information, including the serial number if correct, is stored in the program in a resource of type "Reg." The debug version of Photoshop includes a valid "Reg" resource with registration information, and allows the program to start up fine, as if it were already registered. It shows personalization for Thomas Knoll at Knoll Software. Cool. Photoshop thinks I am Thomas Knoll. Who else would be running it with the debugger enabled?

Things Were Tougher Then

I built Photoshop on a stock Mac IIci with my '040 accelerator removed and the cache card put back in place, to see what the experience would have been to do the build on a high end Macintosh in 1989. The full build, compiling MacApp and the program together, took 14 minutes. If MacApp was already built, it was reduced to 10 minutes.

The debug version of Photoshop starts up normally, and the only immediate differences from the release version are the Debug menu

The screenshot shows the Immunity Debugger interface. The main window displays the CPU registers window for a process named 'Sondra (1:1)'. The 'Debug Window' tab is active, showing a list of registers and their values. The registers are listed in a table with columns for the register name, its value, and a description. The registers shown are: 0: \$EAX, 1: \$ECX, 2: \$EDX, 3: \$EBX, 4: \$ESP, 5: \$EBP, 6: \$ESI, 7: \$EDI, 8: \$EIP, 9: \$EIP, 10: \$EIP, 11: \$EIP, 12: \$EIP, 13: \$EIP, 14: \$EIP, 15: \$EIP. The 'Debug Window' also shows a list of memory addresses and their contents, including: 0: \$EAX, 1: \$ECX, 2: \$EDX, 3: \$EBX, 4: \$ESP, 5: \$EBP, 6: \$ESI, 7: \$EDI, 8: \$EIP, 9: \$EIP, 10: \$EIP, 11: \$EIP, 12: \$EIP, 13: \$EIP, 14: \$EIP, 15: \$EIP. The 'Debug Window' also shows a list of memory addresses and their contents, including: 0: \$EAX, 1: \$ECX, 2: \$EDX, 3: \$EBX, 4: \$ESP, 5: \$EBP, 6: \$ESI, 7: \$EDI, 8: \$EIP, 9: \$EIP, 10: \$EIP, 11: \$EIP, 12: \$EIP, 13: \$EIP, 14: \$EIP, 15: \$EIP.

Some of the commands are pretty obviously useful. The Stack Crawl command prints the name of functions on the stack, and addresses of objects whose methods are on the stack. It is also possible to display a block of memory, a list of recently run subroutines, information about memory allocation

the address of an object, it will print the values stored in all the fields, in hex. Not as useful as you might hope. Turning on trace (T) and starting the program again (G) will create a storm of activity in the Debug Window as subroutines fire off and run their course. By default, the Debug Window only stores 55 lines of text, so it is important to redirect the output to a file. This is probably the most useful general debugging feature.

One of my first lessons learned while reading Photoshop is an appreciation for the power and convenience of the tools we have available for coding today, and the skill of the programmers who made big programs like this without them not too long ago. I'm sure there are a lot more lessons waiting for me, now that the book of Photoshop is open and ready to read. ■

Reprinted with permission of the original author.
First appeared in hn.my/photoshop
(basalgangster.macqui.com)

The StarCraft Path-finding Hack

By PATRICK WYATT

GAME-UNIT PATH-FINDING IS something that most players never notice until it doesn't work quite right, and then that minor issue becomes a rage-inducing, end-of-the-world problem. During the development of StarCraft there were times when path-finding just didn't work at all.

As the development of StarCraft dragged on it seemed like it would never be done: the game was always two months from launch but never seemed to get any closer to the mythical ship date. "Fortunately" — and I use that term advisedly — Blizzard had previous experience shipping games late.

While we always had launch-date "goals" (though "wishes" might be a better term) we tried not to announce them publicly until there was a good chance that the game would be ready at that point. Blizzard's "when it's done" policy for game launch was as much an admission that no one had any idea when we would finish as it was a commitment to releasing quality products.

In any event, towards the end of the project we had a set of problems that prevented launching. Like any game in the latter stages of the development process there were defects galore that needed to be found and repaired and the bug count still numbered in the thousands.

Many of those bugs were trivial and needed only a little attention to fix. Too bad they weren't all like that.

Others, like a multiplayer synchronization bug, would pop up and require dedicated attention from several members of the programming team — sometimes weeks of effort for a single problem. Other game developers have reported similar experiences with their sync bugs: Ages of Empires [hn.my/aoe] and Supreme Commander. [hn.my/scom]

Some bugs were related to the development process itself. The Protoss Carrier regularly lagged behind other units because it had its own way of doing ... everything. At some point in time the code for the Carrier was branched from the main game code and had diverged

beyond any hope of re-integration. Consequently, any time a feature was added for other units, it had to be re-implemented for the Carrier. And any time a bug was fixed for other units, a similar bug would later be found in the Carrier code too, only more devious and difficult to fix.

But the biggest thing holding back StarCraft was unit path-finding.

It wasn't that the path-finding was totally broken; in most cases it worked quite well. But there were enough edge-cases that the game was un-shippable.

Game units would get stuck and stop on the battlefield. Often they would go through elaborate efforts to find paths, inching forward or looping around but not making progress, and sometimes getting wedged and unable to move further. Entire task forces would get bogged down in what looked like the afternoon commute.

The problem was frustrating for players, but it also made the AI weak and consequently made it impossible to balance the missions, wasting design time.

Though I was never a top-tier RTS player, I was good at the game before it launched because I discovered that Goliaths were overpowered to make up for their poor path-finding abilities. Because they were larger than other ground units they needed wider spaces for path-finding, and so by carefully shepherding Goliaths around obstacles I was able to bring their firepower to bear in crucial situations to overcome “macro” players who would otherwise tear me to shreds. Sadly my skills only lasted a short while until the Goliaths were rebalanced. *sigh*

Early path-finding was rough — although there were well-chosen algorithms driving unit movement they were handicapped by some poor development decisions made during the course of the project.

How did we get here?

StarCraft was built on the Warcraft engine, which renders terrain-art using a tile-engine that’s optimized to draw 32×32 pixel square tiles made of 16 8×8 pixel square cells. I architected Warcraft this way because it had worked well for our Super Nintendo and Genesis titles. Those game consoles had hardware support for drawing 8×8 cells, but the behavior was easy to emulate on a PC.

Because the camera perspective of Warcraft I and II were almost top-down, the edges of objects (forests, terrain, buildings) on game maps are either horizontal or vertical, so the render-engine’s representation of the world as a square tile-map is conducive to easy path-finding. In those games, each 32×32 tile was either passable or un-passable. I’ve shown a few of the tile-edges in the

image below in green. Some tiles appeared passable but actually were not; below you can see the barracks building artwork does not fill the 96×96 area it sits on completely, leaving two tiles that appear passable but actually are not (red).



Warcraft 2 map with 32×32 tiles

But along the way the development team switched StarCraft to isometric artwork to make the game more visually attractive. But the underlying terrain engine wasn’t re-engineered to use isometric tiles, only the artwork which was redrawn.

The new camera perspective looked great but in order for path-finding to work properly it was necessary to increase the resolution of the path-finding map: now each 8×8 tile was either passable or un-passable, increasing the size of the path-finding map by a factor of 16. While this finer resolution enabled more units to be squeezed onto the map, it also meant that searching for a path around the map would require substantially more computational effort to search the larger pathing-space.

Path-finding was now more challenging because diagonal edges drawn in the artwork split many of the square tiles unevenly, making it hard to determine whether a tile should be passable or not. Ensuring that all tiles were correctly marked required painstaking effort.

And the StarCraft map editor was horribly difficult to write because of the many edge-cases necessitated by laying down diagonal shapes onto a square tile-map. Writing the specialized “tile-fixup” code necessitated months of programming time.

Unlike Diablo, which used an isometric tile-rendering engine, the StarCraft development team kept the old engine even as new problems with the approach continued to be discovered week by week.

This image shows how a bridge was made up of 8×8 cells; several are shown in green. The almost isometric perspective of the artwork unevenly slices through the cells, leading to a stair-stepped edge along each side of the bridge, as you can see where the red line cuts each tile into an irregular shape.



StarCraft map with 8×8 cells

Because the project was always two months from launch, it was inconceivable that there was enough time to re-engineer the terrain engine to make path-finding easier, so the path-finding code just had to be made to work. To handle all the tricky edge-cases, the pathing code exploded into a gigantic state-machine which encoded all sorts of specialized “get me out of here” hacks.

Rush hour

If there was any one major problem with path-finding it was that harvesting units (Terran SCV, Zerg drone, Protoss probe) would get jammed up trying to harvest crystals or vespene gas (hereafter “minerals”), and they would grind to a halt. While a player was busy managing an attack or constructing a secondary, the harvesters back at the home base would jam up, halting the flow of minerals into the treasury. When next the player looked up, the entire build-queue would have collapsed due to lack of cash.

The basic problem with resource-gathering is that players want to max-out the number of harvesters working on each mineral deposit to maximize their cash flow. Those harvesters are commuting between the minerals and their base so they’re constantly running head-long into other harvesters traveling in the opposite direction. With enough harvesters in a small space, it’s entirely possible that some get jammed in and are unable to move until the mineral deposit is mined out.

How do we get out of here?

I either volunteered or was asked to look into the problem; I just can’t remember after all this time. After studying the path-finding code extensively I realized that there was no way I was smart enough to just “fix the problem.” So I came up with a dirty hack instead.

While programmers can become obsessed with finding the most pure, abstract, clean, even sublime solution to a problem, there are times in a project when a few sacrifices have to be made. If they’re done well no one notices the evil compromises that had to be made, as is true for the hacks written up by Brandon Sheffield in his article Dirty Coding Tricks. [hn.my/dirty]

My idea was simple: whenever harvesters are on their way to get minerals, or when they’re on the way back carrying those minerals, they ignore collisions with other units. By eliminating the inter-unit collision code for the harvesters there is never a rush-hour commute to get jammed up, and harvesters operate efficiently.

It’s possible to notice this behavior by selecting a large group of harvesters who are working a plot of crystals and telling them to halt. They immediately spread out to find tiles that aren’t occupied by other harvesters.

The behavior is obvious if you look, but hidden in plain sight — it doesn’t rise to the level of conscious awareness, though professional-level players and map-makers/modders do notice.

In short, it just works, which is the best kind of hack.

And while there was a lot more work required to finish the game, that hack was what enabled us to launch without massive and time-consuming re-engineering.

The development team was able to work around some of the other path-finding problems and just plain ignore the rest, though the Protoss Dragoon in particular ended up with a bad reputation because, as the largest ground-unit, it frequently failed to path well.

The final result was that path-finding was good enough, and we all learned a hard lesson about hope and wishful thinking as scheduling tools. ■

Patrick Wyatt is a lifelong programmer, game developer, and game-player, and as of 2004, a parent as well. He has worked on many popular games including Warcraft, Diablo, StarCraft, Guild Wars and TERA.

Reprinted with permission of the original author.
First appeared in *hn.my/path* (codeofhonor.com)



MEET MANDRILL

By MailChimp



Mandrill is the fastest way to send transactional, triggered, and personalized emails.
It's also the world's largest species of monkey.

[MANDRILL.COM](https://mandrill.com)

Workflow in Tmux

By EVAN TRAVERS

TMUX HAS THREE levels of hierarchy when it comes to organizing views: sessions, windows, and panes. Sessions are groups of windows, and a window is a layout of panes. Windows and panes are to a certain degree interchangeable as we will see, but sessions are fairly immutable. I use sessions to separate workspaces, almost like the spaces in OSX. Windows and panes I use as is convenient.

I recently have been digging into some of the neater features in tmux's layout system, and here's what I've come up with to help me work harder, better, faster, stronger.

A couple notes: I have set the tmux key to be `ctrl-g`, but you can use whatever you want. For the course of this article, I will use `tmux+[whatever]` to avoid confusion when describing tmux shortcuts. My tmux config can be found here [\[hn.my/tmuxc\]](http://hn.my/tmuxc), and there are some other theme settings here [\[hn.my/tmuxt\]](http://hn.my/tmuxt). When I refer to starting the tmux console, I mean keying in `tmux-:`. Never forget that you always have access to your current shortcuts by typing `tmux-?`.

Sessions

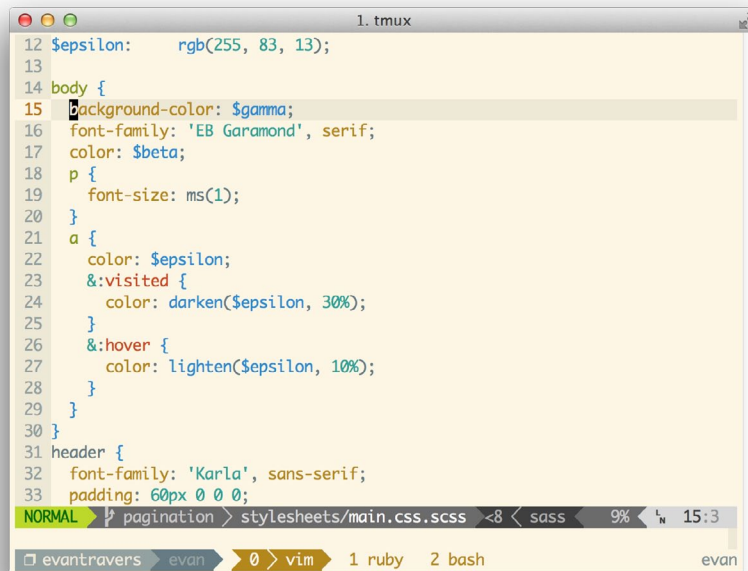
Because I tend to isolate projects to a single session, allowing me to have a complete context switch when needed, I tend to name them the project I am working on, and force their default folder to the project folder. You can switch between sessions using `tmux-s`, rename them using `tmux-$`, and set their default folder by running `tmux set default-folder $(pwd)` inside the session. This last command means that any split or new window inside that session opens in your current folder, which is handy for making a split so you can run a rake task or a script console for your rails project.

I got tired of doing this every time I made a context switch, so I wrote a little script [\[hn.my/tospace\]](http://hn.my/tospace) to do this for me. Super simple, every time I need a new session now I simply `tmux-:` to get the tmux console, type `:new`, then `cd` to the folder I need, and run `tospace` to name it after the project and set the folder defaults.

Windows

Windows in tmux have a name, and a sort number. They live in the bar at the bottom of the screen, ordered by their sort number. You can refer to them in tmux by name or number, which is kind of handy. By default, `tmux-` is rename window, `tmux-.` is move window, and `tmux-[number key]` lets you jump around your first ten windows. I used to think that move window only let you set a new order number for your window, but I have since learned that you can pass windows between sessions fairly easily using the `[session]:[window]` syntax.

If I want to move a window containing my irc session (named `irssi`) from my current session `foo` to the new context session I've created (`bar`), I can either invoke the console and type out `:move-window` or type `tmux-.` to get the same thing, then pass the argument `bar:irssi`. By default, tmux accepts your current window as the argument, but you can move a window you are not currently on by using the `-t -s` arguments to specify the target and source of the move.

A screenshot of a tmux terminal window titled "1. tmux". Inside, a vim editor is open, displaying CSS code. The code includes variables like \$epsilon, \$gamma, and \$beta, and styles for body, p, a, and header elements. The vim status line at the bottom shows "NORMAL" mode, a search for "pagination" in "stylesheets/main.css.scss", and a list of panes: "0 vim", "1 ruby", and "2 bash". The user's name "evantravers" is visible in the bottom left corner of the terminal frame.

```
12 $epsilon:    rgb(255, 83, 13);
13
14 body {
15   background-color: $gamma;
16   font-family: 'EB Garamond', serif;
17   color: $beta;
18   p {
19     font-size: ms(1);
20   }
21   a {
22     color: $epsilon;
23     &:visited {
24       color: darken($epsilon, 30%);
25     }
26     &:hover {
27       color: lighten($epsilon, 10%);
28     }
29   }
30 }
31 header {
32   font-family: 'Karla', sans-serif;
33   padding: 60px 0 0 0;

```

So I had been juggling my irc channel around for a couple of months, before I stumbled on `link-window`. If you invoke the tmux console, type in `link-window`, and you can share a window between two sessions, using the same target/source syntax as `move`. That means I can have the same shell, or the same program shared between multiple sessions. No more juggling the irc window; I can simply have it everywhere.

Panes

Splits are usually the reason people find tmux in the first place, as the version of GNU screen that OSX ships with doesn't do vertical splits for some unknown reason. Horizontal splits are `tmux -"`, and vertical are `tmux -%`. You can rotate your splits around using `tmux -space`, though I've never found a good use for it. More useful is the select layout options available by using `tmux-[meta+1-5]` options, letting you select a variety of layouts. I tend to keep things fairly simple with only one or two splits, so I haven't dived heavily into this yet.

Of course, moving between panes is usually `tmux-[arrow-key]`, but in my polka config I've set it to the vim keybindings, naturally.

One neat trick you can do is pulling a window into your current setup as a pane using `join-pane`. Once again using that `cryptic -t/-s` syntax, you can do some useful things with it. One example I use occasionally is pulling in the window containing the rails server into my dev window so I can access the `binding.pry` session for debugging. For this, I would use `join-pane -t 1` (assuming it's number is 1, I could also use its name). I can even yank it from another session using the `[session]:[number|pane]` syntax. When I'm done, and I want this pane to go back to being another window, you can use `break-pane` to break the pane back out to being a window.

Final thoughts

Because all the tmux commands are also available in the console, and because everything you type in the tmux console is also available by running `tmux [command]`

in your terminal, you can script out workspaces and split setups. I'm considering writing a script to set up my splits and windows for a rails project the way I like them, and building perhaps an argument into the `tspac` script for different layouts. As with all your tools in the command line, think about what you do constantly over and over again, and consider how you could script that into being one step. These tools aren't powerful because you can use the keyboard instead of a mouse — they are powerful because you can combine them in new ways. ■

Evan is a UI developer from Birmingham, AL. Starting from a photography/design background, he fell into becoming software developer, and now enjoys bridging the disparate worlds of engineers and designers. Follow him at [@evantravers](#), or visit his blog at [evantravers.com](#)

Reprinted with permission of the original author.
First appeared in [hn.my/tmux](#) (evantravers.com)

Tips to Accelerate SSL

By GEOFFROY COUPRIE

SSL IS SLOW. These cryptographic algorithms eat the CPU, there is too much traffic, and it is too hard to deploy correctly. SSL is slow. Isn't it?

HELL NO!

SSL looks slow because you haven't tried to optimize it! For that matter, I could say that HTTP is too verbose, XML web services are too verbose, and all this traffic makes the website slow. But SSL can be optimized, as well as everything!

Slow cryptographic algorithms

The cryptographic algorithms used in SSL are not all created equal: some provide better security, some are faster. So, you should choose carefully which algorithm suite you will use.

The default one for Apache 2's SSLCipherSuite directive is:

```
ALL:!ADH:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP
```

You can translate that to a readable list of algorithms with this

```
command: openssl ciphers -v 'ALL:!ADH:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP'
```

Here is the result:

DHE-RSA-AES256-SHA	SSLv3 Kx=DH	Au=RSA	Enc=AES(256)	Mac=SHA1
DHE-DSS-AES256-SHA	SSLv3 Kx=DH	Au=DSS	Enc=AES(256)	Mac=SHA1
AES256-SHA	SSLv3 Kx=RSA	Au=RSA	Enc=AES(256)	Mac=SHA1
DHE-RSA-AES128-SHA	SSLv3 Kx=DH	Au=RSA	Enc=AES(128)	Mac=SHA1
DHE-DSS-AES128-SHA	SSLv3 Kx=DH	Au=DSS	Enc=AES(128)	Mac=SHA1
AES128-SHA	SSLv3 Kx=RSA	Au=RSA	Enc=AES(128)	Mac=SHA1

EDH-RSA-DES-CBC3-SHA	SSLv3 Kx=DH	Au=RSA	Enc=3DES(168)	Mac=SHA1	
EDH-DSS-DES-CBC3-SHA	SSLv3 Kx=DH	Au=DSS	Enc=3DES(168)	Mac=SHA1	
DES-CBC3-SHA	SSLv3 Kx=RSA	Au=RSA	Enc=3DES(168)	Mac=SHA1	
DHE-RSA-SEED-SHA	SSLv3 Kx=DH	Au=RSA	Enc=SEED(128)	Mac=SHA1	
DHE-DSS-SEED-SHA	SSLv3 Kx=DH	Au=DSS	Enc=SEED(128)	Mac=SHA1	
SEED-SHA	SSLv3 Kx=RSA	Au=RSA	Enc=SEED(128)	Mac=SHA1	
RC4-SHA	SSLv3 Kx=RSA	Au=RSA	Enc=RC4(128)	Mac=SHA1	
RC4-MD5	SSLv3 Kx=RSA	Au=RSA	Enc=RC4(128)	Mac=MD5	
EDH-RSA-DES-CBC-SHA	SSLv3 Kx=DH	Au=RSA	Enc=DES(56)	Mac=SHA1	
EDH-DSS-DES-CBC-SHA	SSLv3 Kx=DH	Au=DSS	Enc=DES(56)	Mac=SHA1	
DES-CBC-SHA	SSLv3 Kx=RSA	Au=RSA	Enc=DES(56)	Mac=SHA1	
DES-CBC3-MD5	SSLv2 Kx=RSA	Au=RSA	Enc=3DES(168)	Mac=MD5	
RC2-CBC-MD5	SSLv2 Kx=RSA	Au=RSA	Enc=RC2(128)	Mac=MD5	
RC4-MD5	SSLv2 Kx=RSA	Au=RSA	Enc=RC4(128)	Mac=MD5	
DES-CBC-MD5	SSLv2 Kx=RSA	Au=RSA	Enc=DES(56)	Mac=MD5	
EXP-EDH-RSA-DES-CBC-SHA	SSLv3 Kx=DH(512)	Au=RSA	Enc=DES(40)	Mac=SHA1	export
EXP-EDH-DSS-DES-CBC-SHA	SSLv3 Kx=DH(512)	Au=DSS	Enc=DES(40)	Mac=SHA1	export
EXP-DES-CBC-SHA	SSLv3 Kx=RSA(512)	Au=RSA	Enc=DES(40)	Mac=SHA1	export
EXP-RC2-CBC-MD5	SSLv3 Kx=RSA(512)	Au=RSA	Enc=RC2(40)	Mac=MD5	export
EXP-RC4-MD5	SSLv3 Kx=RSA(512)	Au=RSA	Enc=RC4(40)	Mac=MD5	export
EXP-RC2-CBC-MD5	SSLv2 Kx=RSA(512)	Au=RSA	Enc=RC2(40)	Mac=MD5	export
EXP-RC4-MD5	SSLv2 Kx=RSA(512)	Au=RSA	Enc=RC4(40)	Mac=MD5	export

28 cipher suites, that's a lot! Let's see if we can remove the unsafe ones first! You can see at the end of the list that 7 are marked as "export." That means that they comply with the US cryptographic algorithm exportation policy. Those algorithms are utterly unsafe, and the US abandoned this restriction years ago, so let's remove them:

```
ALL:!ADH:!EXP:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2
```

Now, let's remove the algorithms using plain DES (not 3DES) and RC2:

```
ALL:!ADH:!EXP:!LOW:!RC2:RC4+RSA:+HIGH:+MEDIUM
```

That leaves us with 16 algorithms.

It is time to remove the slow algorithms! To decide, let's use the `openssl speed` command. Use it on your server because you might get different results depending on your hardware. Here is the benchmark on my computer:

```
OpenSSL 0.9.8r 8 Feb 2011
built on: Jun 22 2012
options:bn(64,64) md2(int) rc4(ptr, char) des(idx, cisc, 16, int) aes(partial) blowfish(ptr2)
compiler: -arch x86_64 -fmessage-length=0 -pipe -Wno-trigraphs -fpascal-strings -fasm-blocks
        -O3 -D_REENTRANT -DDSO_DLFCN -DHAVE_DLFCN_H -DL_ENDIAN -DMD32_REG_T=int -DOPENSSL_NO_IDEA
        -DOPENSSL_PIC -DOPENSSL_THREADS -DZLIB -mmacosx-version-min=10.6
available timing options: TIMEB USE_TOD HZ=100 [sysconf value]
timing function used: getrusage
The 'numbers' are in 1000s of bytes per second processed.
```

type	16 bytes	64 bytes	256 bytes	1024 bytes	8192 bytes
md2	2385.73k	4960.60k	6784.54k	7479.39k	7709.04k
mdc2	8978.56k	10020.07k	10327.11k	10363.30k	10382.92k
md4	32786.07k	106466.60k	284815.49k	485957.41k	614100.76k
md5	26936.00k	84091.54k	210543.56k	337615.92k	411102.49k
hmac(md5)	30481.77k	90920.53k	220409.04k	343875.41k	412797.88k
sha1	26321.00k	78241.24k	183521.48k	274885.43k	322359.86k
rmd160	23556.35k	66067.36k	143513.89k	203517.79k	231921.09k
rc4	253076.74k	278841.16k	286491.29k	287414.31k	288675.67k
des cbc	48198.17k	49862.61k	50248.52k	50521.69k	50241.28k
des ede3	18895.61k	19383.95k	19472.94k	19470.03k	19414.27k
idea cbc	0.00	0.00	0.00	0.00	0.00
seed cbc	45698.00k	46178.57k	46041.10k	47332.45k	50548.99k
rc2 cbc	22812.67k	24010.85k	24559.82k	21768.43k	23347.22k
rc5-32/12 cbc	116089.40k	138989.89k	134793.49k	136996.33k	133077.51k
blowfish cbc	65057.64k	68305.24k	72978.75k	70045.37k	71121.64k
cast cbc	48152.49k	51153.19k	51271.61k	51292.70k	47460.88k
aes-128 cbc	99379.58k	103025.53k	103889.18k	104316.39k	97687.94k
aes-192 cbc	82578.60k	85445.04k	85346.23k	84017.31k	87399.06k
aes-256 cbc	70284.17k	72738.06k	73792.20k	74727.31k	75279.22k
camellia-128 cbc	0.00	0.00	0.00	0.00	0.00
camellia-192 cbc	0.00	0.00	0.00	0.00	0.00
camellia-256 cbc	0.00	0.00	0.00	0.00	0.00
sha256	17666.16k	42231.88k	76349.86k	96032.53k	103676.18k
sha512	13047.28k	51985.74k	91311.50k	135024.42k	158613.53k
aes-128 ige	93058.08k	98123.91k	96833.55k	99210.74k	100863.22k
aes-192 ige	76895.61k	84041.67k	78274.36k	79460.06k	77789.76k
aes-256 ige	68410.22k	71244.81k	69274.51k	67296.59k	68206.06k
sign verify sign/s verify/s					
rsa 512 bits	0.000480s	0.000040s	2081.2	24877.7	
rsa 1024 bits	0.002322s	0.000111s	430.6	9013.4	
rsa 2048 bits	0.014092s	0.000372s	71.0	2686.6	
rsa 4096 bits	0.089189s	0.001297s	11.2	771.2	
sign verify sign/s verify/s					
dsa 512 bits	0.000432s	0.000458s	2314.5	2181.2	
dsa 1024 bits	0.001153s	0.001390s	867.6	719.4	
dsa 2048 bits	0.003700s	0.004568s	270.3	218.9	

We can remove the SEED and 3DES suite because they are slower than the other. DES was meant to be fast in hardware implementations, but slow in software, so 3DES (which runs DES three times) is slower. On the contrary, AES can be very fast in software implementations, and even faster if your CPU provides specific instructions for AES. You can see that with a bigger key (and so, better theoretical security), AES gets slower. Depending on the level of security, you may choose different key sizes. According to the key length comparison, 128 might be enough for now. RC4 is a lot faster than other algorithms. AES is considered safer, but the implementation in SSL takes into account the attacks on RC4. So, we will propose this one in priority.

So, here is the new cipher suite:

```
ALL:!ADH:!EXP:!LOW:!RC2:!3DES:!SEED:RC4+RSA:+HIGH:+MEDIUM
```

And the list of ciphers we will use:

DHE-RSA-AES256-SHA	SSLv3 Kx=DH	Au=RSA	Enc=AES(256)	Mac=SHA1
DHE-DSS-AES256-SHA	SSLv3 Kx=DH	Au=DSS	Enc=AES(256)	Mac=SHA1
AES256-SHA	SSLv3 Kx=RSA	Au=RSA	Enc=AES(256)	Mac=SHA1
DHE-RSA-AES128-SHA	SSLv3 Kx=DH	Au=RSA	Enc=AES(128)	Mac=SHA1
DHE-DSS-AES128-SHA	SSLv3 Kx=DH	Au=DSS	Enc=AES(128)	Mac=SHA1
AES128-SHA	SSLv3 Kx=RSA	Au=RSA	Enc=AES(128)	Mac=SHA1
RC4-SHA	SSLv3 Kx=RSA	Au=RSA	Enc=RC4(128)	Mac=SHA1
RC4-MD5	SSLv3 Kx=RSA	Au=RSA	Enc=RC4(128)	Mac=MD5
RC4-MD5	SSLv2 Kx=RSA	Au=RSA	Enc=RC4(128)	Mac=MD5

9 ciphers, that's much more manageable. We could reduce the list further, but it is already in good shape for security and speed. Configure it in Apache with this directive:

```
SSLHonorCipherOrder On
SSLCipherSuite ALL:!ADH:!EXP:!LOW:!RC2:!3DES:!SEED:RC4+RSA:+HIGH:+MEDIUM
```

Configure it in Nginx with this directive:

```
ssl_ciphers ALL:!ADH:!EXP:!LOW:!RC2:!3DES:!SEED:RC4+RSA:+HIGH:+MEDIUM
```

You can also see that the performance of RSA gets worse with key size. With the current security requirements (as of now, January 2013, if you are reading this from the future). You should choose a RSA key of 2048 bits for your certificate because 1024 is not enough anymore, but 4096 is a bit overkill.

Remember, the benchmark depends on the version of OpenSSL, the compilation options and your CPU, so don't forget to test on your server before implementing my recommendations.

Take care of the handshake

The SSL protocol is in fact two protocols (well, three, but the first is not relevant for us): the handshake protocol where the client and the server will verify each other's identity, and the record protocol where data is exchanged.

Here is a representation of the handshake protocol, taken from the TLS 1.0 RFC:

Client		Server
ClientHello	----->	
		ServerHello
		Certificate*
		ServerKeyExchange*
		CertificateRequest*
	<-----	ServerHelloDone
Certificate*		
ClientKeyExchange		
CertificateVerify*		
[ChangeCipherSpec]		
Finished	----->	
		[ChangeCipherSpec]
	<-----	Finished
Application Data	<----->	Application Data

You can see that there are 4 messages exchanged before any real data is sent. If a TCP packet takes 100ms to travel between the browser and your server, the handshake is eating 400ms before the server has sent any data!

And what happens if you make multiple connections to the same server? You do the handshake every time. So, you should activate Keep-Alive. The benefits are even bigger than for plain unencrypted HTTP.

Use this Apache directive to activate Keep-Alive:

```
KeepAlive On
```

Use this nginx directive to activate keep-alive:

```
keepalive_timeout 100
```

Present all the intermediate certification authorities in the handshake

During the handshake, the client will verify that the web server's certificate is signed by a trusted certification authority (CA). Most of the time, there is one or more intermediate certification authorities between the web server and the trusted CA. If the browser doesn't know the intermediate CA, it must look for it and download it. The download URL for the intermediate CA is usually stored in the "Authority information" extension of the certificate, so the browser will find it even if the web server doesn't present the intermediate CA.

This means that if the server doesn't present the intermediate CA certificates, the browser will block the handshake until it has downloaded them and verified that they are valid.

So, if you have intermediate CAs for your server's certificate, configure your webserver to present the full certification chain. With Apache, you just need to concatenate the CA certificates, and indicate them in the configuration with this directive:

```
SSLCertificateChainFile /path/to/certification/chain.pem
```

For nginx, concatenate the CA certificate to the web server certificate and use this directive:

```
ssl_certificate /path/to/certification/chain.pem
```

Activate caching for static assets

By default, the browsers will not cache content served over SSL for security reasons. That means that your static assets (JavaScript, CSS, pictures) will be reloaded on every call. Here is a big performance failure!

The fix for that: set the HTTP header "Cache-Control: public" for the static assets. That way, the browser will cache them. But don't activate it for the sensitive content because it should not be cached on the disk by your browser.

You can use this directive to enable Cache-Control:

```
<filesMatch ".(js|css|png|jpeg|jpg|gif|ico|swf|flv|pdf|zip)$">
Header set Cache-Control "max-age=31536000, public"
</filesMatch>
```

The files will be cached for a year with the max-age option.

For nginx, use this:

```
location ~ \.(js|css|png|jpeg|jpg|gif|ico|swf|flv|pdf|zip)$ {
    expires 24h;
    add_header Cache-Control public;
}
```

Beware of CDN with multiple domains

If you followed the usual performance tips, you already offloaded your static assets (JavaScript, CSS, pictures) to a content delivery network. That is a good idea for a SSL deployment too, BUT, there are caveats:

- Your CDN must have servers accessible over SSL, otherwise you will see the "mixed content" warning
- It must have "Keep-Alive" and "Cache-control: public" activated
- It should serve all your assets from only one domain!

Why the last one? Well, even if multiple domains point to the same IP, the browser will do a new handshake for every domain. So, here, we must go against the common wisdom of separating your assets on multiple domains to profit from the parallelized request in the browser. If all the assets are served from the same domain, there will only be one handshake. It could be fixed to allow multiple domains, but this is beyond the scope of this article. ■

Geoffroy Couprie is a freelance hacker, fixing performance and security problems when he is not shipping weird products. He is now working on *pilotssh.com*, a better way to manage servers from a smartphone.

Reprinted with permission of the original author.
First appeared in *hn.my/acssl* (unhandledexpression.com)

Background image: [flickr.com/photos/nishanthjois/4655396145/](https://www.flickr.com/photos/nishanthjois/4655396145/)

Why Use Make

By MIKE BOSTOCK

I LOVE MAKE. You may think of Make [hn.my/make] as merely a tool for building large binaries or libraries (and it is, almost to a fault), but it's much more than that. **Makefiles are machine-readable documentation that make your workflow reproducible.**

To clarify, this post isn't just about GNU Make; it's about the benefits of capturing workflows via a file-based dependency-tracking build system, including modern alternatives such as Rake and Waf.

To illustrate with a recent example: yesterday Kevin and I needed to update a six-month-old graphic on drought to accompany a new article on thin snowpack in the West. The article was already on the homepage, so the clock was ticking to republish with new data as soon as possible.

Shamefully, I hadn't documented the data-transformation process, and it's painfully easy to forget details over six months: I had a mess of CSV and GeoJSON data files, but not the exact source URL from the NCDC [ncdc.noaa.gov]; I was temporarily confused as to the right Palmer drought metric [hn.my/palmer] (Drought Severity Index or Z Index?) and the corresponding categorical thresholds; finally, I had to resurrect the code to calculate drought coverage area.

Despite these challenges, we republished the updated graphic without too much delay. But I was left thinking how much easier it could have been had I simply recorded the process the first time as a makefile. I could have simply typed `make` in the terminal and be done!

It's Files All The Way Down

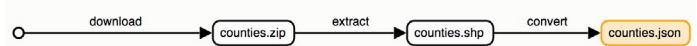
The beauty of Make is that it's simply a rigorous way of recording what you're already doing. It doesn't fundamentally change how you do something, but it encourages you to record each step in the process, enabling you (and your coworkers) to reproduce the entire process later.

The core concept is that generated files depend on other files. When generated files are missing, or when files they depend on have changed, needed files are re-made using a sequence of commands you specify.

Say you're building a choropleth map of unemployment [hn.my/choro] and you need a TopoJSON file of U.S. counties. This file depends on cartographic boundaries published by the U.S. Census Bureau, so your workflow might look like:

1. Download a zip archive from the Census Bureau.
2. Extract the shapefile from the archive.
3. Convert the shapefile to TopoJSON.

As a flow chart:



In a mildly mind-bending maneuver, Make encourages you to express your workflow backwards as dependencies between files, rather than forwards as a sequential recipe. For example, the shapefile depends on the zip archive because you must download the archive before you can extract the shapefile (obviously). So to express your workflow in language that Make understands, consider instead the dependency graph:



This way of thinking can be uncomfortable at first, but it has advantages. Unlike a linear script, a dependency graph is flexible and modular. For example, you can augment the makefile to derive multiple shapefiles from the same zip archive without repeated downloads. Capturing dependencies also begets efficiency: you can remake

generated files with only minimal effort when anything changes. A well-designed makefile allows you to iterate quickly while keeping generated files consistent and up-to-date.

The Syntax Isn't Pretty

The ugly side of Make is its syntax and complexity; the full manual is a whopping 183 pages. Fortunately, you can ignore most of this and start with explicit rules of the following form:

```
targetfile: sourcefile
    command
```

Here `targetfile` is the file you want to generate, `sourcefile` is the file it depends on (is derived from), and `command` is something you run on the terminal to generate the target file. These terms generalize: a source file can itself be a generated file, in turn dependent on other source files; there can be multiple source files or zero source files; and a command can be a sequence of commands or a complex script that you invoke. In Make parlance, source files are referred to as prerequisites, while target files are simply targets.

Here's the rule to download the zip archive from the Census Bureau:

```
counties.zip:
    curl -o counties.zip 'http://www2.census.gov/geo/tiger/GENZ2010/gz_2010_us_050_00_20m.zip'
```

Put this code in a file called `Makefile`, and then run `make` from the same directory. (Note: use tabs rather than spaces to indent the commands in your makefile. Otherwise Make will crash with a cryptic error.) If it worked, you should see a downloaded `counties.zip` in the directory.

You can approximate URL dependencies by checking the Last-Modified header via `curl -I`.

This first rule has no dependencies because it's the first step in the workflow, or equivalently a leaf node in the dependency graph. Although the zip file depends on the Census Bureau's website, and thus can change, Make has no native facility for checking if the contents of a URL have changed, and thus a makefile cannot specify a URL as a prerequisite. As a result, the `counties.zip` file will only be downloaded if it does not yet exist. If the Census Bureau releases new cartographic boundaries, you'll need to delete the previously downloaded zip file before running `make`.

The second rule for creating the shapefile now has a prerequisite: the zip archive.

I preserved the Census Bureau's original verbose file name. You could instead rename files using parameter expansion `[hn.my/paramex]`.

```
gz_2010_us_050_00_20m.shp: counties.zip
    unzip counties.zip
    touch gz_2010_us_050_00_20m.shp
```

This rule also has two commands. First, `unzip` expands the zip archive, producing the desired shapefile and its related files. Second, `touch` sets the modification date of the shapefile to the current time.

The final touch is critical to Make's understanding of the dependency graph. Without it, the modification time of the shapefile will be when it was created by the Census Bureau, rather than when it was extracted. Since the shapefile is apparently older than the zip archive from which it was extracted, Make thinks it needs to be rebuilt — even though it was just made! Fortunately, most programs set the modification dates of their output files to the current time, so you'll probably only need `touch` when using `unzip`.

Lastly to convert to TopoJSON `[hn.my/topojson]`, a rule with one command and one prerequisite:

```
counties.json: gz_2010_us_050_00_20m.shp
    topojson -o counties.json --
counties=gz_2010_us_050_00_20m.shp
```

With these three rules together in a makefile (which you can download), `make counties.json` will perform the necessary steps to produce a U.S. Counties TopoJSON file from scratch.

You can get a lot fancier with your makefiles. For example, pattern rules and automatic variables are useful for generic rules that generate multiple files. But even without these fancy features, hopefully you now have a sense of how Make can capture file-based workflows.

You Should Use Make

Created in 1977, Make has its quirks. But whether you prefer GNU Make or a more recent alternative, consider the benefits of capturing your workflow in a machine-readable format:

- Update any source file, and any dependent files are regenerated with minimal effort. Keep your generated files consistent and up-to-date without memorizing and running your entire workflow by hand. Let the computer work for you!
- Modify any step in the workflow by editing the makefile, and regenerate files with minimal effort. The modular nature of makefiles means that each rule is (typically) self-contained. When starting new projects, recycle rules from earlier projects with a similar workflow.
- Makefiles are testable. Even if you're taking rigorous notes on how you built something, chances are a makefile is more reliable. A makefile won't run if it's missing a step; delete your generated files and rebuild from scratch to test. You can then be confident that you've fully captured your workflow.

To see more real-world examples of makefiles, see my World Atlas [hn.my/worldatlas] and U.S. Atlas [hn.my/usatlas] projects, which contain makefiles for generating TopoJSON from Natural Earth, the National Atlas, the Census Bureau, and other sources. The beauty of the makefile approach is that I don't need gigabytes of source data in my git repositories (Make will download them as needed), and the makefile is infinitely more customizable than pre-generating a fixed set of files. If you want to customize how the files are generated, or even just use the makefile to learn by example, it's all there.

So do your future self and coworkers a favor, and use Make! ■

Mike Bostock is a graphics editor for The New York Times and the author of D3.js, a popular open-source library for visualizing data using web standards. Previously, Mike was a visualization scientist for Square and a computer science PhD student at Stanford University.

Reprinted with permission of the original author.
First appeared in hn.my/usemake (bost.ocks.org)

Where and When Did the Symbols “+” and “−” Originate?

By MARIO LIVIO

THE SYMBOLS FOR the arithmetic operations of addition (plus; “+”) and subtraction (minus; “−”) are so common today we hardly ever think about the fact that they didn’t always exist. In fact, someone first had to invent these symbols (or at least other ones that later evolved into the current form), and some time surely had to pass before the symbols were universally adopted. When I started looking into the history of these signs, I discovered to my surprise that they did not have their origin in antiquity. Much of what we know is based on an impressively comprehensive and still unsurpassed body of research from 1928–1929 entitled *History of Mathematical Notations* by the Swiss-American historian of mathematics, Florian Cajori (1859–1930).

The ancient Greeks expressed addition mostly by juxtaposition, but sporadically used the slash symbol “/” for addition and a

semi-elliptical curve for subtraction. In the famous Egyptian Ahmes papyrus, a pair of legs walking forward marked addition, and walking away subtraction. The Hindus, like the Greeks, usually had no mark for addition, except that “*yu*” was used in the Bakhshali manuscript Arithmetic (which probably dates to the third or fourth century). Towards the end of the fifteenth century, the French mathematician Chuquet (in 1484) and the Italian Pacioli (in 1494) used “**p**” or “*p*” (indicating plus) for addition and “**m**” or “*m*” (indicating minus) for subtraction.

There is little doubt that our + sign has its roots in one of the forms of the word “et,” meaning “and” in Latin. The first person who may have used the + sign as an abbreviation for et was the astronomer Nicole d’Oresme (author of *The Book of the Sky and the World*) in the middle of the fourteenth century. A manuscript from 1417 also has the + symbol (although the

downward stroke is not quite vertical) as a descendent of one of the forms of et.

The origin of the − sign is much less clear, and speculations range all the way from hieroglyphic or Alexandrian grammar ancestry, to a bar symbol used by merchants to separate the tare from the total weight of goods.

The first use of the modern algebraic sign − appears in a German algebra manuscript from 1481 that was found in the Dresden Library. In a Latin manuscript from the same period (also in the Dresden Library), both symbols + and − appear. Johannes Widman is known to have examined and annotated both of those manuscripts. In 1489, in Leipzig, he published the first printed book (*Mercantile Arithmetic*) in which the two signs + and − occurred (Figure 1). The fact that Widman used the symbols as if they were generally known points to the possibility that they were derived

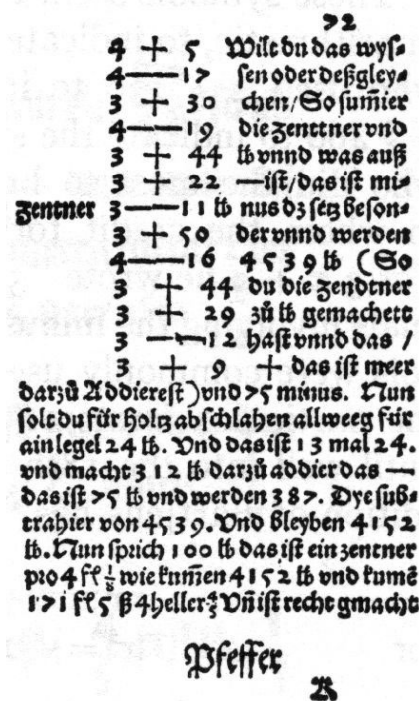


Figure 1. The first use of the + and – symbols in print in Johannes Widman's *Behende und Lubche Rechenung auff allen Kauffmanschafft*, Augsburg edition of 1526.

from merchants' practices. An anonymous manuscript — probably written around the same time — also used the same symbols, and it provided input for two additional books published in 1518 and 1525.

In Italy, the symbols + and – were adopted by the astronomer Christopher Clavius (a German who lived in Rome), the mathematicians Gloriosi, and Cavalieri at the beginning of the seventeenth century.

The first appearance of + and – in English was in the 1551 book on algebra, *The Whetstone of Witte* by the Oxford mathematician Robert Recorde, who also introduced the equal sign as the rather longer than today's symbol “=.” In describing the plus and minus signs Recorde wrote: “There be other 2 signes in often use of which the first is made thus + and betokeneth more: the other is thus made – and betokeneth lesse.”

As a historical curiosity, I should note that even once adopted, not everybody used precisely the same symbol for +. Widman himself introduced it as a Greek cross + (the sign we use today), with the horizontal stroke sometimes a bit longer than the vertical one. Mathematicians such as Recorde, Harriot and Descartes used this form. Others (e.g., Hume, Huygens, and Fermat) used the Latin cross “†,” sometimes placed horizontally, with the crossbar at one end or the other. Finally, a few (e.g., De Hortege, Halley) used the more ornamental form “✕.”

The practices of denoting subtraction were somewhat less fanciful, but perhaps more confusing (to us at least), since instead of the simple –, German, Swiss, and Dutch books sometimes used the symbol “÷,” which we now use for division. A few seventeenth century books (e.g., by Descartes and by Mersenne) used two dots “..” or three dots “...” for subtraction.

Overall, what is perhaps most impressive in this story is the fact that symbols which first appeared in print only about five hundred years ago have become part of what is perhaps the most universal “language.” Whether you do science or finances, in Kentucky or in Siberia, you know precisely what these symbols signify. ■

Mario Livio is an astrophysicist and an author of popular science books. His research interests range from extrasolar planets to supernova explosions and cosmology. He is also passionate about art.

Reprinted with permission of the original author.
 First appeared in *hn.my/symbol* (blogs.stsci.edu/livio)

What It's Really Like Working With Steve Jobs



By GLENN REID

PEOPLE WHO WORKED with Steve Jobs (I'll call him Steve) usually don't talk about it. It's kind of an unwritten rule, partly because he was obsessive about his privacy.

I think that has all changed now, but I'm not exactly sure. I am at least sure that my phone won't ring if I say something about the experience. And I feel compelled to do just that, because there is so much written about Steve, and so few who have actually seen him work. I was one of those people.

And I am becoming aware that lots of people are claiming, in one way or another, to have been one of those people. "I worked with Steve Jobs" can mean, "I saw him in the elevator once when I was at a meeting at Apple," or "I worked at Apple during those years, and saw him around campus, although I never actually spoke to him." I actually worked with the guy, and I'm realizing that perhaps I worked with him more closely than almost anyone (save Avie and the many who were in his inner circle for

the whole duration of course) — because I worked on products that he cared deeply about.

First, some background. I worked at Adobe Systems in 1985, one of the first handful of people at the company. I was employee #40. After about 5 years, I was searching for something new to do, and got interested in NeXT, because they embraced PostScript (an Adobe technology) and were UNIX-based, two things that I was good at. Being young and brash, I wrote an email message directly to Steve, suggesting that I was just the right guy to work there. In 1991, I started work at NeXT, as Product Manager for Interpersonal Computing. It was the internet, before there was much of an internet. We called it Interpersonal Computing, but nobody paid attention until 5 years later when the WWW became more mainstream. I reported directly to Steve, in his capacity as "acting VP of Marketing," which was a lifelong title for him.

I left NeXT to start a company to build software for NeXT computers, RightBrain Software. We built an amazing page layout app called PasteUp, ran 2-page spread ads in NeXTWORLD magazine, and had a good old time, except we didn't sell a lot of software, so I went off to do other things for a while.

Many years later, when NeXT acquired Apple for negative \$400M, I was recruited by Steve's right hand man to come in to build iMovie 1.0, in large part because I knew a lot about NeXTSTEP, the technology which was to become MacOS X, and because I think Steve liked PasteUp and liked me and thought I could get it done (we were done ahead of schedule, as it turned out).

“Steve would draw a quick vision on the whiteboard, we’d go work on it for a while, bring it back, find out the ways in which it sucked, and we’d iterate, again and again and again.”

I can still remember some of those early meetings, with 3 or 4 of us in a locked room somewhere on Apple campus, with a lot of whiteboards, talking about what iMovie should be (and should not be). It was as pure as pure gets, in terms of building software. Steve would draw a quick vision on the whiteboard, we’d go work on it for a while, bring it back, find out the ways in which it sucked, and we’d iterate, again and again and again. That’s how it always went. Iteration. It’s the key to design, really. Just keep improving it until you have to ship it.

There were only 3 of us on the team, growing to 4 within the year, with no marketing and very little infrastructure around us. There was paper over the internal windows to keep other Apple employees from knowing what we were doing. Our component in Radar, the bug-tracking database, was called “Tax Department” so nobody would be curious about it. We sat in the same hallway as the Tax Department, actually, and our Senior VP was

in charge of Service and Support at the time. Truly a stealth project. There were maybe only 5-10 people in the whole company who knew what we were doing.

When we were done, and the iMac DV shipped with iMovie built in (I think it was October or November of 1998), the world changed, for everybody. Jeff Goldblum appeared in TV ad spots, showing off iMovie. The idea of “personal digital media” was born. This was Steve’s vision, and why he put together the iMac DV, with Firewire and iMovie. We called it the Digital Hub strategy internally, to encourage you to put lots of personal digital media on your home computer. It grew quickly from movies to include photos and music (iTunes was repackaged SoundJam, acquired from Casady and Greene in 2000). Before then, very few people had any personal photos, or music, or home movies on their computers.

Over the ensuing 5 years or so, we built several versions of iMovie and several versions of iPhoto, which came out a couple years after iMovie, but along the same track. Toward the end of my time at Apple, we had standing meetings, once a week, for about 3 or 4 hours, in the Board Room at Apple, to go through what were known internally as the “iApps” — iMovie, iPhoto, iTunes, and later iDVD. Over the course of some years, that’s a lot of CEO hours devoted to the details of some software apps — and that was just the part that we saw. I’m sure there were similar meetings for OS X, the Pro apps, the hardware, and everything else that was going on.

Now let me back up a bit.

Steve Jobs was passionate, as everyone knows. What he was passionate about was, I think, quite simple: he liked to build products. I do, too. This we had in common. It is a process which requires understanding the parameters, the goals, and the gives and takes. Stretch what's possible, use technologies that are good, rein it in when the time comes, polish it and ship it. It's a kind of horse sense, maybe a bit like building houses, where you just kind of know how to do it...or you don't. Steve did.

Not only did he know and love product engineering, it's all he really wanted to do. He told me once that part of the reason he wanted to be CEO was so that nobody could tell him that he wasn't allowed to participate in the nitty-gritty of product design. He was right there in the middle of it. All of it. As a team member, not as CEO. He quietly left his CEO hat by the door, and collaborated with us. He was basically the Product Manager for all of the products I worked on, even though there eventually were other people with that title, who usually weren't allowed in the room.

One of the things about designing products that can come up is Ego, or Being Right, or whatever that is called. I'm not sure how this evolved, but when I worked with Steve on product design, there was kind of an approach we took, unconsciously, which I characterize in my mind as a "cauldron." There might be 3 or 4 or even 10 of us in the room, looking at, say, an iteration of iPhoto. Ideas would come forth, suggestions, observations, whatever. We would "throw them into the cauldron," and stir it, and

soon nobody remembered exactly whose ideas were which. This let us make a great soup, a great potion, without worrying about who had what idea. This was critically important, in retrospect, to decouple the CEO from the ideas. If an idea was good, we'd all eventually agree on it, and if it was bad, it just kind of sank to the bottom of the pot. We didn't really remember whose ideas were which — it just didn't matter. Until, of course, the patent attorneys came around and asked, but that's a whole other story.

The Steve that I worked with loved product design, and he loved consumer products, and iMovie and iPhoto were two of the biggest consumer apps ever developed from scratch at Apple, or NeXT, or anywhere else, perhaps. So I think that in some very real sense, I had a better understanding of Steve and how he worked, and what motivated him, than almost anyone in the world. It sounds kind of self-serving to say this, but he and I were a lot alike in that way, and in that process. It was a true give and take, a true collaboration with everyone in the room. Most people never saw that process, and those who did never talk about it. I am privileged to have been there.

I guess I have this to say about it: it wasn't magic, it was hard work, thoughtful design, and constant iteration. Doing the best we knew how with what was available, shaping each release into a credible, solid, useful, product, as simple and direct as we could make it. And we shipped those products, most importantly.

I am off doing other things now, again, but it's still Product Design, and I still love it. That is what I remember most about Steve, that he simply loved designing and shipping products. Again, and again, and again. None of the magic that has become Apple would have ever happened if he were simply a CEO. Steve's magic recipe was that he was a product designer at his core, who was smart enough to know that the best way to design products was to have the magic wand of CEO in one of your hands. He was compelling and powerful and all that, but I think that having once had the reigns of power wrestled away from him, he realized that it was important not to let that happen again, lest he not be allowed to be a Product Manager any more. ■

Glenn Reid is a long-time hacker and entrepreneur. He was the original author of iMovie and iPhoto at Apple, employee #40 at Adobe, and CEO of multiple startups, including Five Across, which was acquired by Cisco in 2007.

Reprinted with permission of the original author.
First appeared in hn.my/sjobs (inventor-labs.com)

Illustration by Jimmy "good work media"
[goodworkmedia.se]



HOW WOULD YOU FIX FINANCE



Engineers rebuilding the infrastructure
that powers finance. → careers.addepar.com

stripe

Accept payments online.