Ideal OS: Rebooting the Desktop Operating System Experience

Modern desktop operating systems are anything but.

Contents

Description

Modern desktop operating systems are ***bloated, slow, and layered with legacy cruft*** that still functions only thanks to Moore's Law. ***Innovation in desktop operating systems stopped about 15 years ago*** and the major players are unlikely to heavily invest in them again. We can and ***should start over from scratch***, learning the lessons of the past.

-This idea was sourced from a blog post and used with permission from its original creator, Josh Marinacci -

No Innovation

Innovation in desktop operating systems is essentially dead. One could argue that it ended sometime in the mid-90s, or even in the 80s with the release of the Mac, but clearly all progress stopped after the smartphone revolution.

Mac OS

Mac OS X was once a shining beacon of new features, with every release showing profound progress and invention. Quartz 2D! Expose! System wide device syncing! Widgets! Today, however Apple puts little effort into their desktop operating system besides changing the theme every now and then and increasing hooks to their mobile devices.

Apple's newest version of Mac OS X (now renamed macOS in honor of where they were two decades ago) is called High Sierra. What are banner features that we are eagerly awaiting this fall? A new filesystem and a new video encoding format. Really, that's it? Oh, and they added editing to Photos, which was already there in iPhotos but removed during the upgrade and they will block autoplay videos now in Safari.

Apple is the most valuable company in the world and this is the best they can do? Desktop UX just isn't a priority for them.

Microsoft Windows

On the Windows side there has been a flurry of activity as Microsoft tried to reinvent the desktop as a touch operating system for tablets and phones. This was a disaster that they are still recovering from. In the process of this shift they didn't add any features that actually helped desktop users, though they did spend an absurd amount of money creating a custom background image.

Instead of improving the desktop UX they focused on adding new application models with more and more layers on top of the old code. Incidentally, Windows can *still* run applications from the early 90s.

CMD.exe, the terminal program which essentially still lets you run DOS apps was only replaced in 2016. And the biggest new feature of the latest Windows 10 release? They added a Linux subsystem. More layers piled on top.

X Windows

X Windows has improved even less than the other two desktop OSes. In fact, it's the very model of non-change. People were complaining about it in the early 90s. I'm glad that I can reskin my GUI toolkit, but how about a system wide clipboard that holds more than one item at a time? That hasn't changed since the 80s!

X added compositing window managers in the mid-2000s, but due to legacy issues it can't be used for anything beyond sliding your windows around.

Work Stations?

Fundamentally desktop operating systems became easier to use as they were adopted by the mass market; then the mass market moved to smartphones and all interest in improving the desktop interface stopped.

I can't blame Apple and Microsoft (and now Google) for this. 3 billion smartphones replaced every two years is a far bigger market than a few hundred million desktops and laptops replaced every five.

I think we need to take back the desktop operating system experience. We used to call these things *workstations*. If the desktop is freed from being the OS for the masses, then it can go back to being an OS for work.

Things we don't have in 2017

Why can I dock and undock tabs in my web browser or in my file manager, but I can't dock a tab between the two apps? There is no technical reason why this shouldn't be possible. Application windows are just bitmaps at the end of the day, but the OS guys haven't built it because it's not a priority.

Why can't I have a file in two places at once on my filesystem? Why is it fundamentally hierarchical? Why can I sort by tags and metadata? Database filesystems have existed for decades. Microsoft tried to build it with WinFS, but that was removed from Vista before it shipped thanks to internal conflicts. BeOS shipped it twenty years ago. Why don't we have them in our desktop OSes today?

Any web app can be zoomed. I can just hit *command +* and the text grows bigger. Everything inside the window automatically rescales to adapt. Why don't my native apps do that? Why can't I have one window big and another small? Or even scale them automatically as I move between the windows? All of these things are trivial to do with a compositing window manager, which has been commonplace for well over a decade.

Limited Interaction

My computer has a mouse, keyboard, tilt sensors, light sensors, two cameras, three microphones, and an array of bluetooth accessories; yet only the first two are used as general input devices. Why can't I speak commands to my computer or have it watch as I draw signs in the air, or better yet watch as I work to tell me when I'm tired and should take a break.

Why can't my computer watch my eyes to see what I'm reading, or scan what I'm holding in my hands using some of that cool AR technology coming to my phone. Some of these features do exist as isolated applications, but they aren't system wide and they aren't programmable.

Why can't my Macbook Pro use Bluetooth for talking to interesting HID devices instead of syncing to my Apple Wait. Oh wait, my Mac *can't* sync to my Apple Watch. Another place where my desktop plays second fiddle to my phone.

My can't my computer use anything other than the screen for output? My new Razor laptop has an RGB light embedded under every key, and yet it's only used for waves of color. How about we use these LEDs for something useful! (via Bjorn Stahl, I think).

Application Silos

Essentially every application on my computer is a silo. Each application has its own part of the filesystem, its own config system, and its own preferences, database, file formats and search algorithms. Even its own set of key bindings. This is an incredible amount of duplicated effort.

More importantly, the lack of communication between applications makes it very difficult to get them to coordinate. The founding principle of Unix was small tools that work together, but X Windows doesn't enable that at all.

What can we get rid of that doesn't work well?

- **Traditional filesystems** are hierarchical, slow to search, and don't natively store all of the metadata we need.
- **All IPC.** There are too many ways for programs to communicate. Pipes, sockets, shared memory, RPC, kernel calls, drag and drop, cut and paste.
- **Command line interfaces** don't fit modern application usage. We simply can't do everything with pure text. I'd like to pipe my Skype call to a video analysis service while I'm chatting, but I can't really run a video stream through awk or sed.
- **Window Managers** on traditional desktops are not context or content aware, and they are not controlable by other programs.
- **Native Applications** are heavy weight, take a long time to develop and very siloed.

So what does that leave us with? Not much. We have a kernel and device drivers. We can keep a reliable filesystem but it won't be exposed to end users or applications.

Document Database (improvement)

Start with a system wide document database. Wouldn't it be easier to build a new email client if the database was already built for you? The UI would only be a few lines of code. In fact, many common applications are just text editors combined with data queries. Consider iTunes, Address Book, Calendar, Alarms, Messaging, Evernote, Todo list, Bookmarks, Browser History, Password Database, and Photo manager. All of these are backed by their own unique datastore. Such wasted effort, and a block to interoperability.

BeOS proved that a database filesystem could really function and provide incredible advantages. We need to bring it back.

A document database filesystem has many advantages over a traditional one. Not only can 'files' exist in more than one place, and they become easily searchable, having a guaranteed performant database makes app building far easier.

Consider iTunes. iTunes stores the actual mp3 files on disk, but all metadata in a private database. Having two sources of truth causes endless problems. If you add a new song on disk you must manually tell iTunes to rescan it. If you want to make a program that works with the song database you have to reverse engineer iTunes DB format, and pray that Apple doesn't change it. All of these problems go away with a single system wide database.

Message Bus (improvement)

A message bus will be the only kind of IPC. We get rid of sockets, files, pipes, ioctrls, shared memory, semaphores, and everything else. All communication is through a message bus. This gives us one place to manage security and enables lots of interesting features through clever proxying.

In reality we probably would continue to have some of these available as options for apps that need it, like sockets for a webbrowser, but all communication to the system and between apps should be messages.

Compositor (improvement)

Now we can add in a compositing window manager that really just moves surfaces around in 3D, transforms coordinates, and is controlled through messages. Most of what a typical window manager does, like arranging windows, overlaying notifications, and determining which window has focus; can actually be done by other programs who just send messages to the compositor to do the real work.

This means the compositor is heavily integrated with the graphics driver, which is essential to making such a system fast. Below is the diagram for Wayland, the compositor which will eventually become the default for desktop Linux.

Applications would do their drawing by requesting a graphics surface from the compositor. When they finish their drawing and are ready to update they just send a message saying: please repaint me. In practice we'd probably have a few types of surfaces for 2d and 3d graphics, and possibly raw framebuffers. The important thing is that at the end of the day it is the compositor which controls what ends up on the real screen, and when. If one app goes crazy the compositor can throttle it's repaints to ensure the rest of the system stays live.

Apps become modules (improvement)

All applications become small modules that communicate through the message bus for everything. **Everything**. No more file system access. No hardware access. Everything is a message.

If you want to play an mp3 you send a play message to an mp3 service. You draw by having the compositor do it for you. This separation makes the system far more secure. In Linux terms each app would be completely isolated through user permissions and chroot, or perhaps all the way to docker containers or virtual machines. There's a lot of details to work out, but this is very doable today.

Module apps would be far easier to write than today. If the database is the single source of truth then a lot of the general work of copying data in and out of memory can go away. In the music player example, instead of the search field loading up data and filtering it to show the list, the search field just specifies a query. The list is then bound to this query and data automatically flows in. If another application adds a song to the database that matches the search query, the music player UI will automatically update. This is all without any extra work from the app developer. Live queries make so many things easier and more robust.

Rebuild the Apps (improvement)

From this base we should be able to build everything we need. However, this also means we *have* to rebuild everything from scratch. Higher level constructs built on top of the database would make many applications a lot easier to rebuild. Let's look at some examples.

Email. If we separate the standard email client into GUI and networking modules, which communicate solely through messages, then building a client becomes a lot easier. The GUI doesn't have to know anything about GMail vs Yahoo mail or how to process SMTP error messages. It simply looks for documents with type 'email' in them. When the GUI wants to send a message it marks an email with the property *outgoing=true.* A headless module will listing for outgoing emails and do the actual STMP processing.

Splitting the email app into component makes replacing one part far easier. You could build a new email frontend in an afternoon without having to rebuild the networking parts. You could build a spam detector that has *no UI at all*, it just listens for incoming messages, processes them, and marks the bad ones with a spam tag. It doesn't know or care how spam is visualized. It just does one thing well.

Email filters could do other interesting things. Perhaps you send an email to your bot with the command 'play beatles'. A tiny module looks for this incoming email, sends another message to the mp3 module for playing the music, then marks the email as deleted.

Once everything becomes a database query the entire system becomes more flexible and hackable.

The command line (improvement)

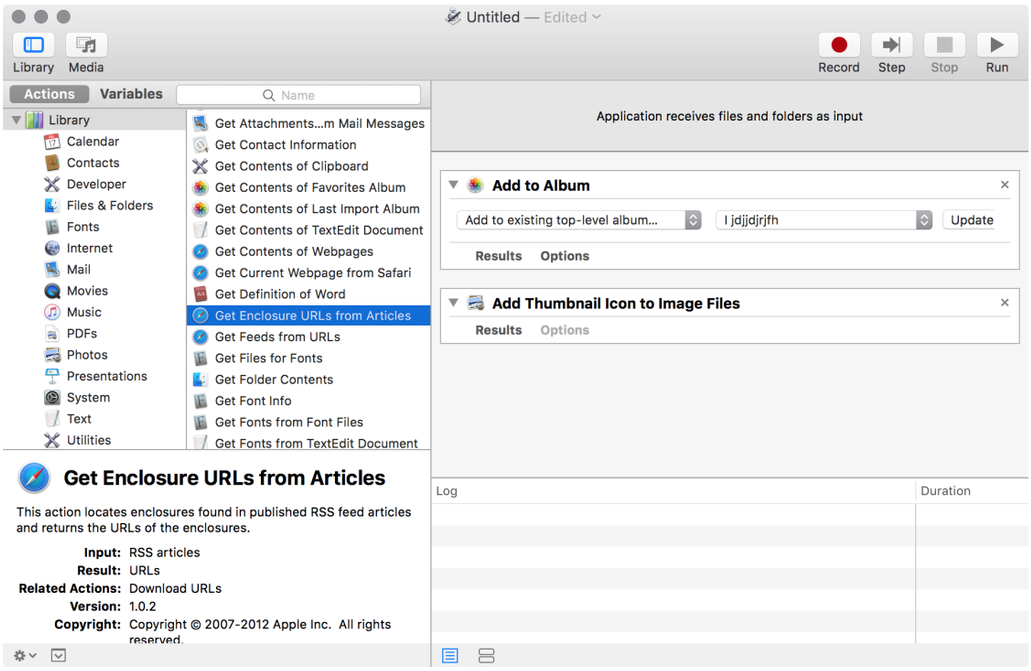
I know I said we would get rid of the commandline before, but I take that back. I really like the commandline as an interface sometimes, it's the pure text nature that bothers me. Instead of chaining CLI apps together with text streams we need something richer, like serialized object streams (think JSON but more efficient). Then we start getting some real power.

Consider the following tasks:

- I want to use my laptop as an amplified microphone. I speak into it and the sound comes out of a bluetooth speaker on the other side of the room.
- Whenever I tweet something with the hashtag #mom I want a copy sent, by email, to my mother.
- I want to use my iphone sitting on a stand made of legos as microscope. It streams to my laptop, which has controls to record, pause, zoom, and rebroadcast as a live stream to youtube.
- I want to make a simple bayesian filter which detects emails from my power company, adds the tag 'utility', logs into the website, fetches the current bill amount and due date, and adds an entry to my calendar.

Each of these tasks is conceptually simple but think of how much code you would have to write to actually make this work today. With a CLI built on object streams each of these examples could become a one or two line script.

We could do complex operations like 'find all photos taken in the last four years within 50 miles of Yosemite, and that have a star rating of 3 or higher, resize them to be 1000px on the longest size, then upload them to a Flickr album called "Best of Yosemite", and link to the album on Facebook'. This could be done with built in tools, no custom coding required, just by combining a few primitives.



Apple actually built such a system. It's called Automator. You can visually create powerful workflows. They never promote it, have started deprecating the Applescript bindings which make it work underneath, and recently laid off or transferred all of the members of the Automator team. Ah well.

System Side Semantic Keybindings (improvement)

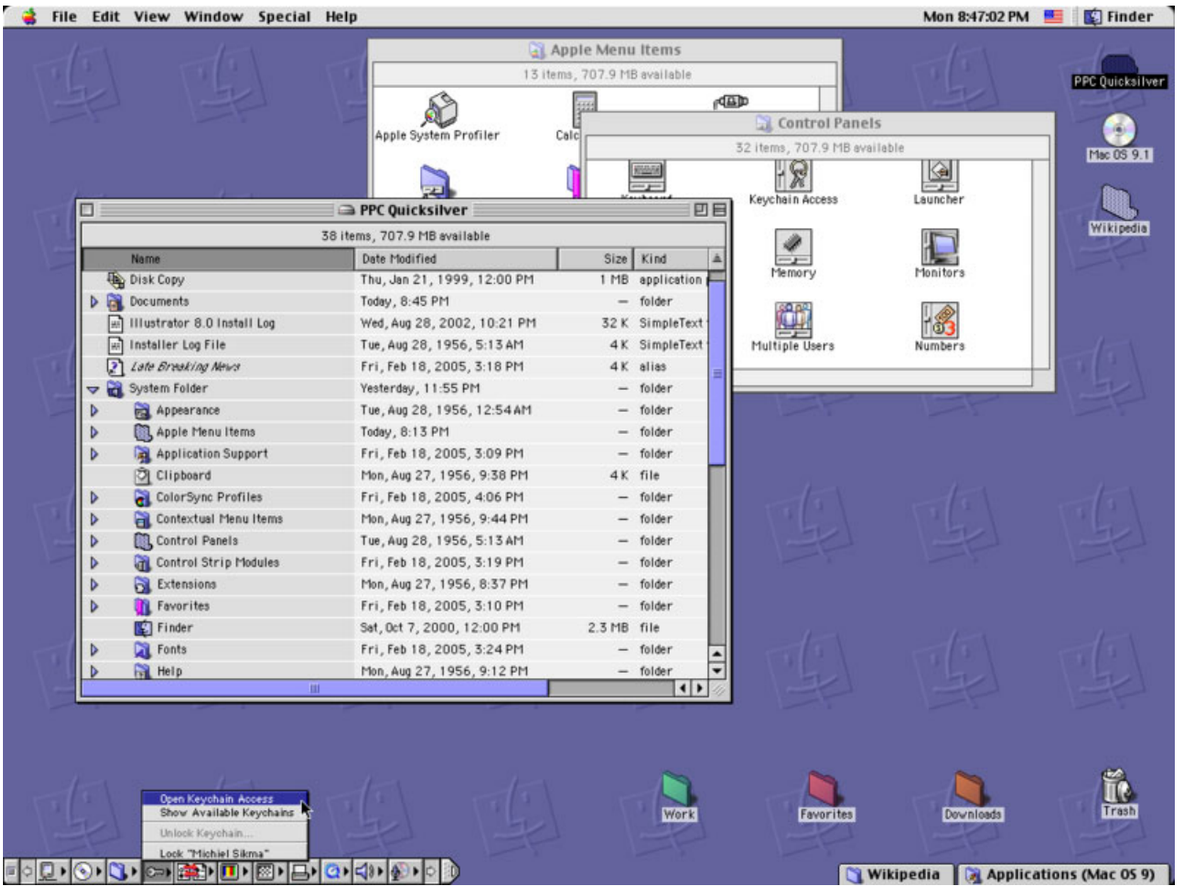
Services are available system wide. This means we could have a keybinding service which gives the user one place to set up keybindings. It also means we could have a richer sense of what a keybinding is. Instead of mapping a key to a function in a particular program, a key binding maps a key combo to a command message. All applications that work on documents could have a 'save' or 'new' command. The keybinding service would be responsible for turning a control-S into the save command. I call these semantic keybindings.

Semantic keybindings would also make it a lot easier to support alternate forms of input. Suppose you built a weird Arduino button thing that speaks every time you mash a button. You wouldn't need to write any custom code. Just make the arduino send in a new keypress event, then map it to a play audio message in the bindings editor. Turn a digital pot into a custom scroll wheel. Your UI is now fully hackable.

I need to do some more research in this area, but I suspect semantic keybindings would make screen readers and other accessibility software easier to build.

Windows (improvement)

In our new OS every window is tab dockable. Or side dockable. Or something else. The apps don't care. We have a lot of freedom to explore here.



The old MacOS 8 had a form of tabbed windows, at least for Finder windows, where you could dock them at the bottom of the screen for easy access. Another cool thing that was left behind in the transition to Mac OS X

In the screenshot below the user is lifting the edge of a window up to see what's underneath. That's super cool!

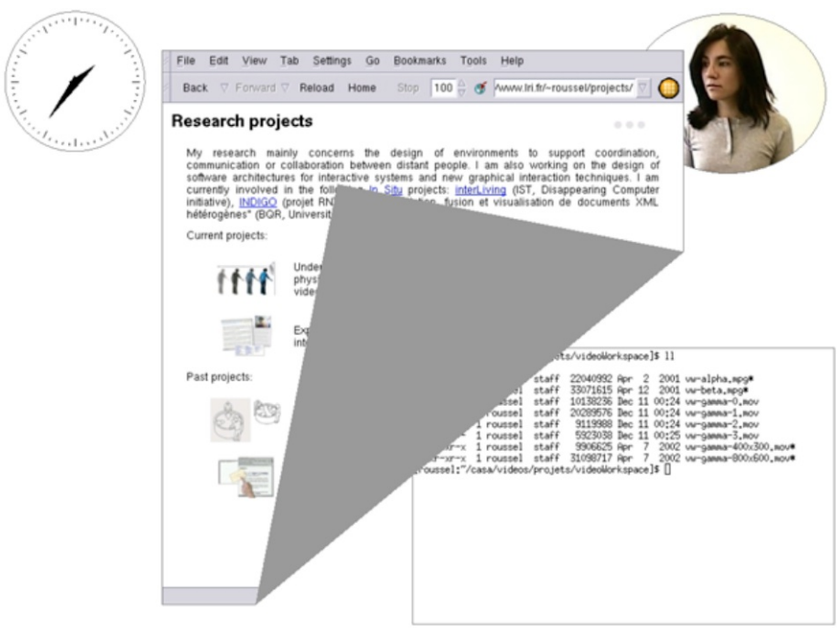


Figure 4: Examples of window shape transformations using texture mapping.

This is an example from *Ametista: a mini-toolkit for exploring new window management techniques* , a research paper by Nicolas Roussel.

Since the system fully controls the environment of all apps, it could enforce security restrictions and show that to the user. A trusted system app could have a green border. A new app downloaded from the internet would have a red border. An app with an unknown origin could have a black border, or just not be shown at all. Many kinds of spoofing attacks become impossible.

Smart copy and paste (improvement)

When you copy in one window then shift to another, the computer knows that you just copied something. It can now use this knowledge to do something useful, like automatically shift the first window to the side, but still visible, and render the selected text in glowing green. This keeps the user's mind on the task at hand. When the user pastes into a new window we could show the green text actually leap from one window to another.

But why stop there. Let's make a clipboard that can hold more than one item at at time. We have gigs of RAM. Let's use it. When I copy something why do I have to remember what I copied before I paste it? The clipboard isn't actually visible anywhere. Let's fix that.

The clipboard should be visible on screen as some sort of a shelf that shows the recent items I've copied. I can visit three webpages, copy each url to the clipboard, then go back to my document and paste all three at once.

This clipboard viewer can let me scroll through my entire clipping history. I could search and filter it with tags. I could 'pin' my favorites for use later.

Classic macOS actually had an amazing tool built into it called [name], but it was dropped in the shift to OS X. We had the future decades ago! Let's bring it back.

Working Sets (improvement)

And finally we get to what I think is the most powerful metaphor change in our new Ideal OS. In the new system all applications are tiny isolated things which only know what the system tells them. If the treat the database as the single source of truth, and the database itself is versioned, and our window manager is fully hackable... then some really interesting things become possible.

Usually I have a split between personal files and files for work. I tend to use separate folders, accounts, and sometimes separate computers. In the Ideal OS my files could actually be separated by the OS. I could have one screen up with my home email, and another screen with my work email. These are the exact same app, just initialized with different query settings.

When I open a file browser on the home screen it only shows files designated as home projects. If I create a document on my work screen the new file is automatically tagged as being work only. Managing all of this is trivial; just extra fields in the database.

Researchers at Georgia Tech actually built a version of this in their research paper: *Giornata: Re-Envisioning the Desktop Metaphor to Support Activities in Knowledge Work*.

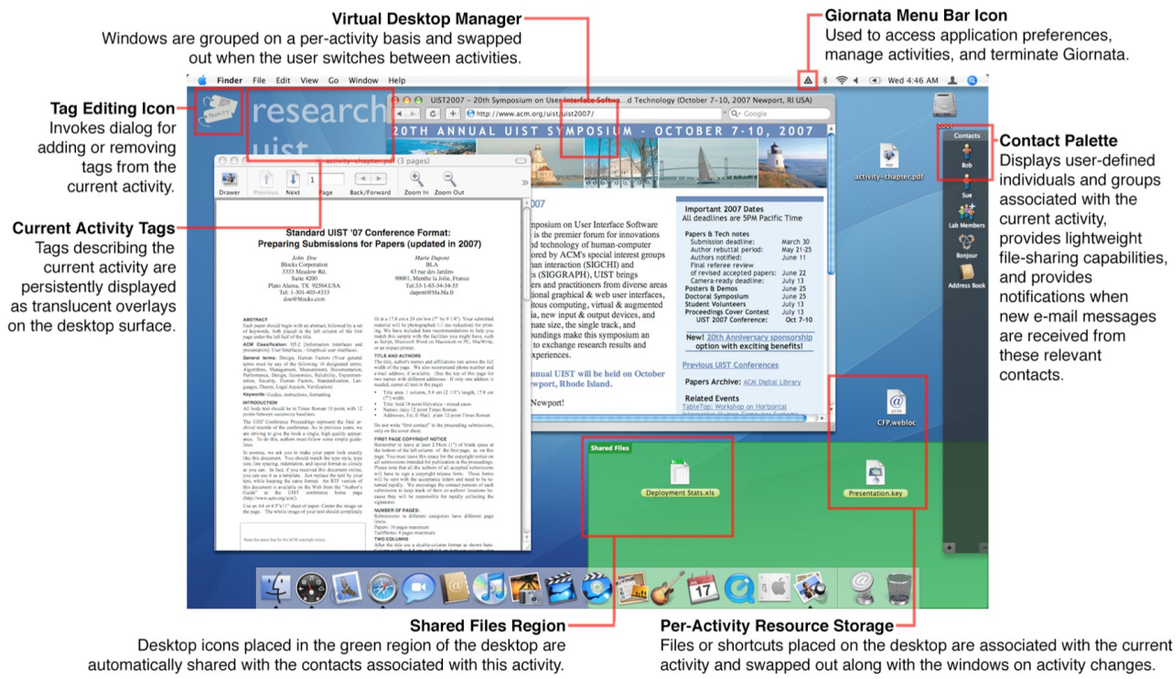


Figure 1. The Giornata user interface. In this screenshot, the user is engaged in an activity of preparing a conference submission. There are several tags (including “research” and “uist”), two open windows, four files (two of them shared), two colleagues, and one group currently associated with this activity.

Now let's take things one step further. If everything is versioned, even GUI settings and window positioned (since it's all stored in the database), I could take a snapshot of a screen. This would store the current state of everything, even my keybindings. I can continue working, but if I want I could **rollback** to that snapshot. Or I could view the old snapshot and restore it to a new screen. Now I have essentially created a 'template' that I can use over and over whenever I start a new project. This template can contain anything I want: email settings, chat history, todos, code, issue windows, or even a github view.

Now we can essentially treat all state in the computer like a github repo, with the ability to fork the state of the entire system. I think this would be huge. People would exchange useful workspaces online much as they do with Docker images. People could tweak their workflows add useful scripts embedded into the workspaces. The possibilities really are amazing.

Problem

"Modern" Desktop Operating Systems are Bloated

Consider the Raspberry Pi. For 35 dollars I can buy an amazing computer with four CPU cores, each running over a *gigahertz*. It also has a 3d accelerator, a gig of RAM, and built in wifi & bluetooth & ethernet. For 35 bucks! And yet, for many of the tasks I want to do with it, this Raspberry Pi is no better than the 66 *megahertz* computer I used in college.

In fact, in some cases it's worse. It took tremendous effort to get 3D accelerated Doom to work inside of X windows in the mid 2000s, something that was trivial with mid-1990s Microsoft Windows.

Processing ran for the first time on a Raspberry Pi with hardware acceleration, just a couple of years ago. And it was possible only thanks to a completely custom X windows video driver. This driver is still experimental and unreleased, *five years* after the Raspberry Pi shipped.

Despite the problems of X-Windows, the Raspberry Pi has a surpisingly powerful GPU that can do all sorts of things, but only once we get X windows out of the way. Here's another example. Atom is one of the most popular editors today. Developers love it because it has oodles of plugins, but let us consider how it's written. Atom uses Electron, which is essentially an entire webbrowser married to a NodeJS runtime. That's two Javascript engines bundled up into a single app. Electron apps use browser drawing apis which delegate to native drawing apis, which then delegate to the GPU (if you're luck) for the actual drawing. So many *layers*.

For a long time Atom couldn't open a file larger than 2 megabytes because scrolling would be to slow. They solved it by writing the buffer implementation in C++, essentially removing one layer of indirection.

Even fairly simple apps are pretty complex these days. An email app is conceptually simple. It should just be a few database queries, a text editor, and a module that knows how to communicate with IMAP and SMTP servers. Yet writing a new email client is very difficult and consumes many megabytes on disk, so few people do it. And if you wanted to modify your email client, or at least the one above (Mail.app, the default client for Mac), there is no clean way to extend it. There are no plugins. There is no extension API. This is the result of many layers of cruft and bloat.

No Innovation

Innovation in desktop operating systems is essentially dead. One could argue that it ended sometime in the mid-90s, or even in the 80s with the release of the Mac, but clearly all progress stopped after the smartphone revolution.