Riccardo Antonelli

Raytracing a Black Hole

HACKERMONTHLY

Issue 59 April 2015

Curator
Lim Cheng Soon

Contributors
Riccardo Antonelli
Evan Todd
Vince Buffalo
Stephen Wyatt Bush
Josh Johnson
Rob Conery
Eve Fisher
Matt Welsh

Proofreader
Emily Griffin

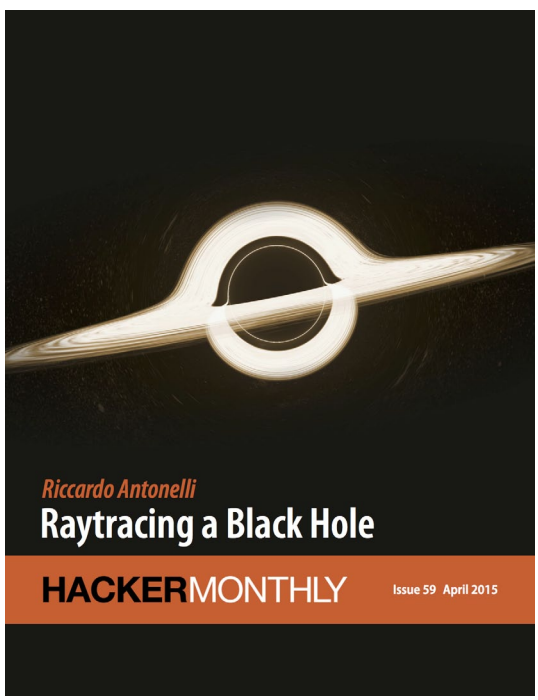
Printer
Blurb

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.



Contents

For links to Hacker News dicussions, visit *hackermonthly.com/issue-59*

Raytracing a Black Hole

By RICCARDO ANTONELLI

It's now clear I'm on a Black Hole binge (I can stop when I want, by the way). They're endlessly fascinating. My recent interest was in particular focused on simulating visualizations of the Schwarzschild geometry. I was preoccupied by the problem of generating a decent accurate representation of how the curvature of such a spacetime affects the appearance of the sky (since photons from distance sources ride along geodesics bent by the Black Hole), for the purpose of creating an interactive simulation. This was the result (it runs in your browser). The trick was, of course, to pre-calculate as much as possible about the deflection of light rays. It worked ok-ish, but the simulation is of course very lacking in features, since it's not actually doing any raytracing (for the laymen: reconstructing the whereabouts of light rays incoming in the camera back in time) on its own.

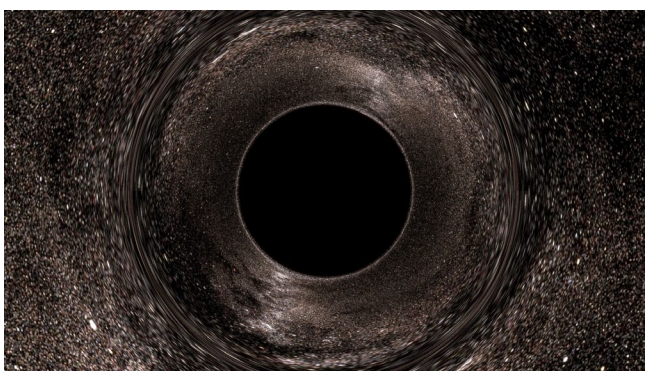
This project, instead, aims to shatter these shortcoming by ditching efficiency/interactivity in the most naive way: it's a full CPU raytracer, taking all the time it needs to render pictures. The image above was rendered with this program. It took 5 minutes on my laptop.

This is neither anything new nor is it any better than how it has been done before. It's just really fun for me. I'm writing this to share not only end-results, but also the process of building these pictures, with a discussion/explanation of the physics involved and the implementation. Ideally, this could be of inspiration or guidance to people with a similar intent.

A bit of pseudo-Riemannian optics

The shadow

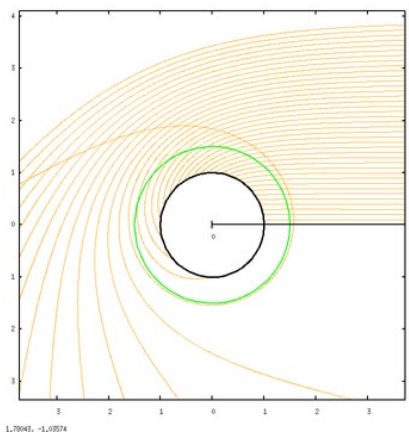
If you have already tried my live applet [hn.my/bhapplet], you should be familiar with this view:



You shouldn't have problems making out the salient feature of the image, namely the black disk and the weird distortion ring.

It's often pointed out that it's incorrect to say that the black disk is the event horizon. In fact, it's incorrect to say that a region of an image is an object. These are images

of things. Now, it's true that there will be rays that, when backtraced from your eye, will end up in the event horizon. These will be black pixels, since no photon could ever have followed that path going forward, from inside the black hole to your eye. This black disk is thus very clearly the image of the event horizon, in the sense that if you draw (in the far past) something right above the horizon, outside observers will be able to see it right on that black disk (we will actually perform this experiment later). This black region is also called "shadow" of the BH in some publications.



What's interesting to note, however, is that this is at the same time the image of the photon sphere. The gnuplot graph above depicts geodesics of incoming photons from infinity (looking at the BH from far away zooming in) along with the EH (black) and the PS (green). The photon sphere is 32 times the event horizon (in Schwarzschild r) and is the location where circular orbits of light around the BH are allowed (though unstable). In the graph, identify rays that fall to their death and those who get only scattered (and thus end up on another point on the celestial sphere). You see that absorbed rays are those arriving with an impact parameter of less than ~ 2.5 radii. This is the apparent radius of the black disk, and it's significantly larger than both the EH and the PS.

Anyways, the relevant trivia here is this:

> *A light ray infalling in the photon sphere in free fall will also reach the event horizon.*

This implies that the image of the photon sphere is included in that of the horizon. However, since the horizon is very clearly inside the photon sphere, the image of the former must also be a subset of that of the latter. Then the two images should coincide.

Why should you care that the black disk is also the image of the PS? Because it means that the edge of the black disk is populated by photons that skim the photon sphere. A pixel right outside the black disk corresponds to a photon that (when tracing backwards) spirals into the photon sphere, getting closer and closer to the unstable circular orbit, winding many times (the closer you look, the more it winds), then spiraling out (since the orbit is unstable) and escaping to infinity.

This behavior will produce an interesting optical phenomenon and is basically getting close to a separatrix in a dynamical system. In the limit, a ray thrown exactly on the edge will spiral in forever, getting closer and closer to the photon sphere circular orbit.

The effect on the celestial sphere

I'm not going to focus a lot on this, because this was the main goal of the live applet, and you can get a much better idea of the distortions induced on the sky through that (which also includes a UV grid option so the distortion is clearer).

Just a couple of things about the Einstein ring. The Einstein ring is distinguishable as an optical feature because it is the image of a single point, namely that on the sky directly opposite the observer. The ring forms at the view angle where rays from the observer are bent parallel. Outside of it, rays are not bent enough and remain divergent; inside, they are bent too much and converge and in fact can go backwards, or even wind around multiple times, as we've seen.

But then, think about this: if we get close enough to the black disk, light rays should be able to wind around once and then walk away parallel. There we should see a secondary Einstein ring, and, in fact, rings of any order (any number of windings). Also, there should be "odd" rings in between where light rays are bent parallel but directed towards the viewer. This infinite series of rings is there, but it's absolutely invisible in this image (in fact, in most of them), as they are very close to the disk edge.

The distortion of the Event Horizon

In this new image, there are a couple of things that have changed. First of all, this was rendered at a higher resolution and with filtering for the background, so as to be more readable. Then, I've zoomed in on the hole (haven't gotten closer, we're still at ~ 10 radii, just zoomed in). But most importantly, I have drawn a grid on the horizon.

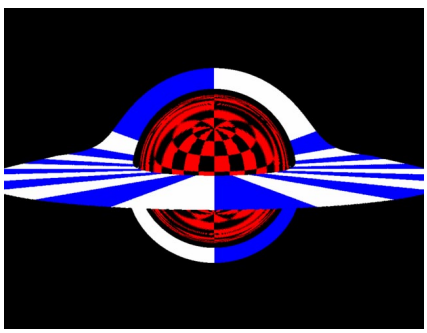
The horizon is "just a sphere." Technically, it does not work like a standard Riemannian sphere with a spatial metric. The horizon is light-like! A pictorial way of saying this is that it's going outwards at the speed of light. However, in Schwarzschild coordinates, it's still a $r = 1$ surface, and we can use ϕ and θ as longitude and latitude. So it's possible to draw a coordinate grid in a canonical way. Then what you're seeing is how that grid would look.

The grid allows us to take note of a peculiar fact we could have also deduced by analyzing the photon scattering/absorption graph above:

> *The whole surface of the horizon is visible at the same time, from any point*

This is very interesting. When you look at a stationary sphere in standard flat spacetime, you can see at most 50% of its surface at any given time (less if you're closer, because of perspective). The horizon, instead, is all visible simultaneously, mapped in the black disk: notice in particular the North and South poles. However, while the surface of the EH is all there, it doesn't cover all of the black disk: if you zoomed in on the edge, you'd see that this image of the EH ends before the shadow ends. Namely you'll find a ring, very close to the outside edge, but not equal, which is an image of the point opposite the observer and delimits this "first" image of the EH inside. So what's in between this ring and the actual edge? I haven't yet bothered making a zoom to show this, but there's another whole image of the event horizon squeezed in there. And then another, and then another, ad infinitum. There are infinite concentric images of the whole horizon, squeezed on the shadow.

Adding an accretion disk



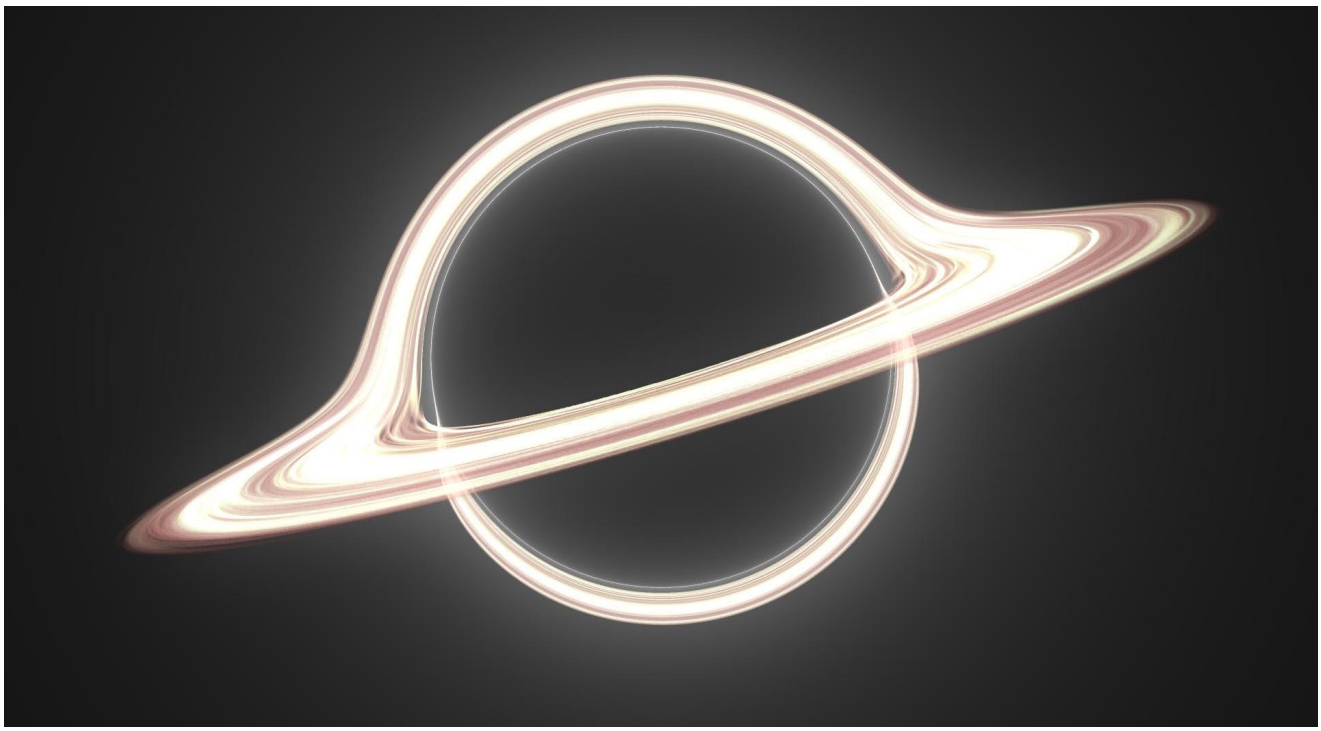
What modern black hole rendering would it be without an accretion disk? While it's certainly debatable whether Nolan's Interstellar was actually watchable, not to mention accurate, we can certainly thank the blockbuster for popularizing the particular way the image of an accretion disk is distorted. Here we have an infinitely thin, flat, horizontal accretion disk extending from the photon sphere (this is very unrealistic, orbits below 3rS are unstable. More below) to 4 radii, colored checkered white and blue on the top and white and green on the bottom. It is evident, with this coloring, that we've encountered another case of seeing 100% of something at the same time.

For this image, I moved the observer up a bit, so he can take a better look at the disk. You can see two main images of the disk, one of the upper face, and one, inside, of the lower. The blue image has the far section of the upper disk distorted to arch above the shadow of the BH. This happens because a ray pointing right above the black hole is bent down to meet the upper surface of the disk behind the hole, opposite the observer.

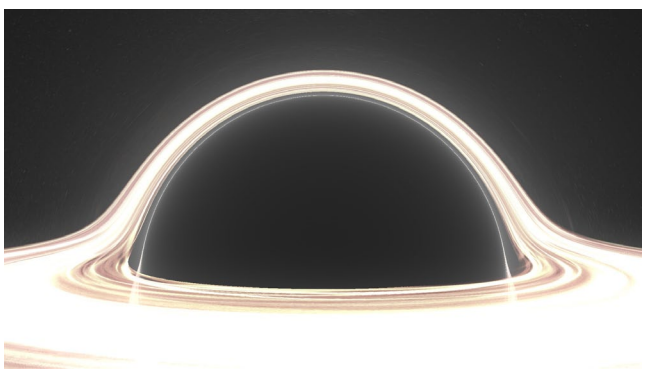
This also explains the very existence of the green image: rays going below are bent to meet the lower surface, still behind the hole. The green image, if you look closely, extends all around the shadow, but it's much thinner in the upper section. This corresponds to light rays that go above the BH, are bent into an almost full circle around the hole, and hit the lower surface in the front section of the disk.

Of course, it's easy to deduce that there is an infinite series of accretion disk images, getting very quickly thinner and closer to the edge. The next-order image, in blue, is already very thin but faintly visible in the lower portion of the edge.

Enough science.
Enough with the informative pixelated 90's uni mainframe renderings with garish colors.
Here are some "pop" renders.

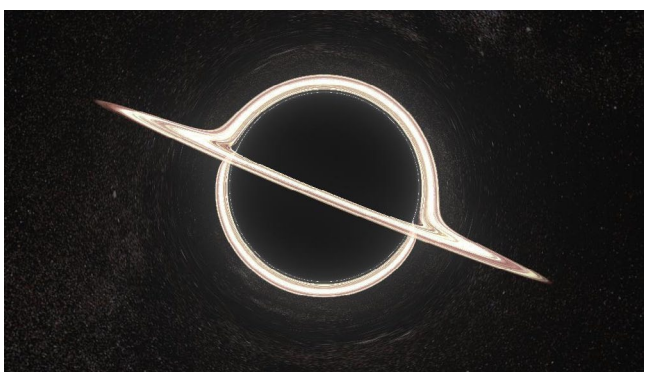


This image was rendered by *reddit.com/u/dfzxh* with x4 supersampling.

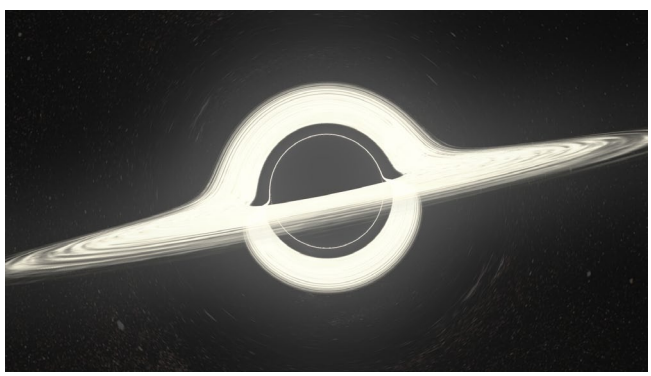


A closer look.

Zoom on the ring images.



Iconic "ring of light" effect when looking from the equatorial plane.



If you download the program, this is the current default scene.

Yeah, they're nothing special. Not an artist here. Let's get back temporarily to the science: the "Zoom on the ring" image, the one that doesn't seem to make any sense, is actually very precious. It's a zoom on the region between the upper edge of the black disk and the main ("first blue") image of the accretion disk. The observer is placed on the outer rim of the accretion disk itself and zooms in on this detail. The goal was to image as many orders of rings as possible. Three orders are visible: the lighter zone at the top is just the lower rim of the first image of the top-far surface of the disk. The strip at the bottom, below a calm sea of outstretched stars, is the superior part of the second image, the "first green" one, of the bottom-front of the disk. At the very bottom is a thin line of light not more than a pixel wide, glued to the black disk of the photon sphere. This is mainly the third image, the "second blue": it's the image again of the top-far surface, but after the light has completed an additional winding around the black hole. Merged with it, but increasingly thin, are all subsequent higher-order images. Ok, this is something worthy of `<blockquote>` tags:

> *There are infinite images of both the upper and lower surface of the accretion disk, and they all show the whole surface simultaneously. Moreover, except for the very first, these images don't pass in front of the black disk nor each other, and are thus "concentric."*

Marvelous.

Realistic accretion disc

The accretion disc in the renders above is cartoony. It's just a disc with a stupid texture splattered on it. What happens when in the visual appearance of the disc we include physics-aware information? What happens when we include redshift from orbital motion, for example?

A popular model for an accretion disc is an infinitely thin disc of matter in almost circular orbit, starting at the ISCO (Innermost Stable Circular Orbit, $3r_s$), with a power law temperature profile $T \sim r^{-a}$. I'll use the extremely simple

$$T \sim r^{-3/4}$$

Which is most definitely not ok in GR for realistic fluids, but it'll do.

A free parameter now is the overall scale for the temperatures, for example the temperature at the ISCO. This temperature is immense for most black holes. We're talking hundreds of millions of Kelvin; it's difficult to imagine any human artefact that could survive being exposed to the light (peaking in X-rays) of a disc at those temperatures, let alone capture anything meaningful on a CCD. We then really have to tone it down. Apparently supermassive black holes are colder, but not enough. We need to pull it down to around 10,000 K at the ISCO for us to be able to see anything. This is highly inaccurate, but it's all I can do.

We need to ask ourselves two questions. One: what color is a blackbody at that temperature? Two: how bright is it? Formally, the answer to those two questions is in the scalar product of the functions describing R,G,B channels with the black body spectrum. In practice, one uses some approximations.

For color, this formula by Tanner Helland [hn.my/tanner] is accurate and efficient, but it involves numerous conditionals which are not feasible with my raytracing setup (see below for details). The fastest way is to use a lookup texture:

This texture is one of many goodies from Mitchell Charity's "What color is a blackbody?". (For reference, it corresponds to white-point E).

This runs from 1000 K to 30,000 K, higher temperatures are basically the same shade of blue. Since there is an immense difference in brightness between temperatures, this texture cannot and does not encode brightness; rather, the colors are normalized. It is our duty to compute relative brightness and multiply. We can use an analytic formula for that. If we assume that the visible spectrum is very narrow, then the total visible intensity is proportional to the blackbody spectrum itself:

$$\frac{1}{\lambda^5}\frac{1}{\exp(\frac{hc}{\lambda k_B T}) - 1}$$

Where I got rid of stupid overall constants (we're going to rescale brightness anyway to see anything). We can just plug in λ roughly in the visible spectrum range and we get that brightness is proportional to:

$$(e^{\frac{29622.4\text{K}}{T}} - 1)^{-1}$$

That's easy enough. As a check, we note that relative intensity quickly drops to zero when T goes to zero, and is only linear in T as T goes to infinity.

Redshift

I discussed the orbital speeds in the Schwarzschild geometry in the explanation for the live applet. To compute redshift, we use the special-relativistic redshift formula:

$$(1+z)_{\text{Doppler}} = \frac{1-\beta\cos(\theta)}{\sqrt{1-\beta^2}}$$

Where cos(θ) is the cosine of the angle between the ray direction when it's emitted by the disc and the disc local velocity, all computed in the local inertial frame associated with the Schwarzschild coordinates. This formula is correct in this context because of the equivalence principle.

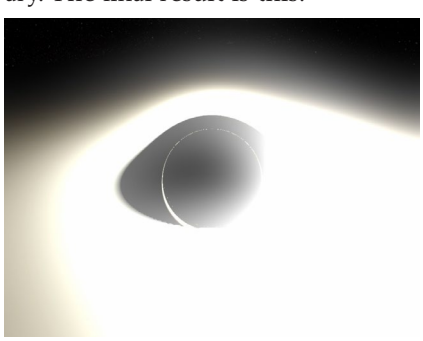
This is to be multiplied with the gravitational redshift factor:

$$(1+z)_{\text{Gravitational}} = (1-r^{-1})^{-1/2}$$

This factor does not depend on the path of the light ray, only on the emission radius, because the Schwarzschild geometry is stationary.

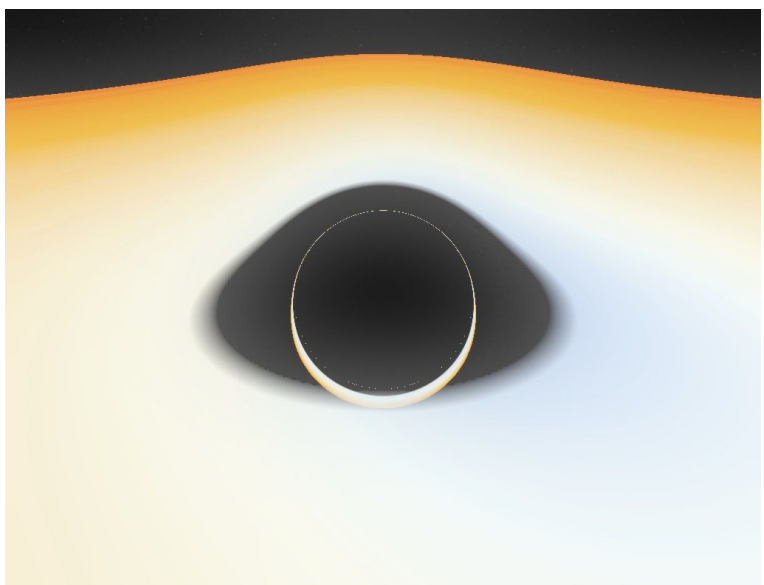
This also means that the contribution to gravitational redshift due to the position of the observer is constant over the whole field of view. All our image gets a constant overall blueshift because we're deep in the hole's well. This effect therefore is just applying a tint over our image, and we ignore it.

We also neglect redshift from observer motion, because our observer is Schwarzschild-stationary. The final result is this:



As you can see, most of the disc is completely white, because it saturates the color channels. If I scale down those channels to fit in the 0.0-1.0 range, the outer parts of the disk become faint or black. The growth in brightness is too large for us to appreciate. I've tried to depict it in post processing through a bloom effect to make really bright parts bleed instead of just clip, but it's hardly sufficient.

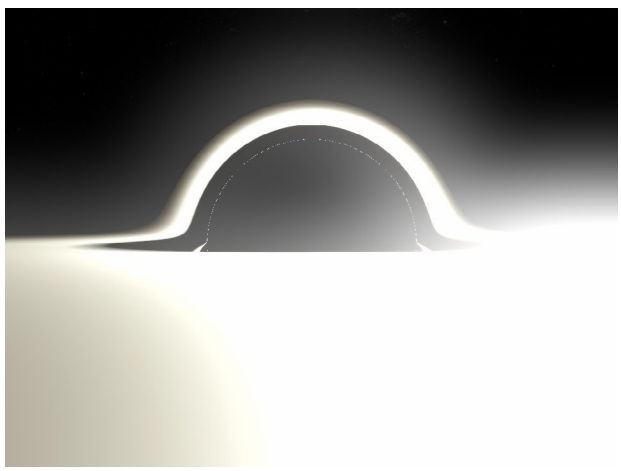
Quite a confusing picture. Here's a picture with the intensity ignored, so you can appreciate the colors:



These are at a smaller resolution because they take so long to render on my laptop (square roots are bad, kids).

Anyways, it looks thousands of time less scenographic than the other renders (mostly because the inner edge of the disk is already far away enough from the EH that lensing looks quite underwhelming), but at least it's accurate, if you managed to find a 10,000 K black hole and some really good sunglasses, that is.

Another shot from a closer distance. I tweaked saturation unnaturally up so you can tell better:



Writing a black hole raytracer

There is very obviously a massive difference between understanding the qualitative aspects of black hole optics and building a numerical integrator that spits out 1080p ok-ish wallpaper material. Last time I neglected the aspect of explaining my thought processes in coding and I put up a really messy git repo. I want to go into a little more detail now and will try to maintain the code tidier and commented.

I don't want this raytracer to be good, solid, fast. I want it to be easy and hackable, so that people can be inspired by it, may it be because they see potential for improvement or because it's so sh***y it makes them want to make their own. So here's a quick walkthrough of the algorithms and implementation.

The "magic" potential

So, General Relativity, right. Easy. Take the Schwarzschild metric, find the Christoffel symbols, find their derivative, write down the geodesic equation, change to some Cartesian coordinates to avoid endless suffering, get an immense multiline ODE, and integrate. That's pretty much it.

Just kidding. Of course there's a trick.

If you remember last time, I derived the following equation for the orbit of a massless particle in its orbital plane in a Schwarzschild geometry ($u = 1/r$):

$$u''(\phi) + u = \frac{3}{2}u^3$$

The trick is to recognize that this is in the form of a Binet equation. If you have an absolutely massive and Newtonian particle in a Newtonian central potential:

$$\frac{d^2}{dt^2}\vec{x} = \frac{1}{m}F(r)$$

Then the particle will obviously move in its orbital plane and will satisfy the Binet equation for u(ϕ):

$$u'' + u = -\frac{1}{mh^2u^2}F(u)$$

Where the prime is d/dϕ, m is the mass and h is the angular momentum per unit mass. This is an equation for the orbit, not an equation of motion. It does not tell you anything about u(t) or ϕ(t), just the relationship between u and ϕ.

Let's pause a moment to ponder what this is actually telling us. It says that if we were to evolve a hypothetical mechanical system of a particle under a certain central force, its trajectory will be a solution to the Binet equation. Then the mechanical system becomes a computational tool to solve the latter.

What I propose here is exactly this. We put m=1 and take the (unphysical, whatever) simple system of a point particle in this specific force field:

$$\vec{F}(r) = -\frac{3}{2}h^2\frac{\hat{r}}{r^5}$$

Where h is some constant, and we integrate that numerically — it's very easy. Then the solution $\vec{x}(T)$, where T is the abstract time coordinate for this system, is actually a parametrization of the unique solution for the corresponding Binet equation, which is exactly the geodesic equation.

So we solve Newton's equation in Cartesian coordinates, which is the easiest thing ever; I use the leapfrog method instead of RK4 because it's simple, reversible and preserves the constants of motion. (I now switched to Runge-Kutta to be able to increase step size and reduce render times, but with the future possibility of leaving the choice of integration method to the user). Then what I obtain is just the actual light-like geodesic with T a parameter running along it (distinct from both Schwarzschild t and proper time, that doesn't exist).

This is much better than the previous method, which worked with polar coordinates in the orbital plane. This is very efficient computationally.

Raytracing in numpy

If you take a look at the source tree, you'll find is not much of a tree. It's just a Python script. The horror! Why would anyone write a raytracer in Python? Python loops are notoriously heavy, which is mostly (but not completely) a deal breaker. The point here is that we're doing the computations in numpy and calculating everything in parallel. This is why this program won't show you progressively the parts of the image it has already rendered: it's raytracing them all at the same time.

One basically starts by creating an array of initial conditions. For example, a `(numPixel,3)` array with view 3-vector corresponding to every pixel of the image (numPixel is image width * image height). Then every computation one would do for a single ray, one does in `(numPixel, ...)`-shaped arrays. Every quantity is actually an array. Since operations on numpy arrays are very fast and everything is statically typed (hope I'm not saying anything stupid right now) this should be fast enough. Maybe not C, but fast-ish.

At the same time, we have the versatility and clarity of Python.

This method is basically horrible for standard raytracing, where objects have diffuse, reflective, refractive components and illumination conditions are important. Selectively reflecting parts of an array of rays, for example, is a nightmare; taking track of Booleans or loop indices requires numerous masks, and loops cannot be broken out of. However, this is not our case: the only objects in our scene are exclusively emissive: the sky, the incandescent accretion disk, the pitch black event horizon, and the bright dust. These are unaffected by incoming light on the object, and light itself passes through them undisturbed, at most reduced in intensity. This leads us to our color determination algorithm:

Color blending

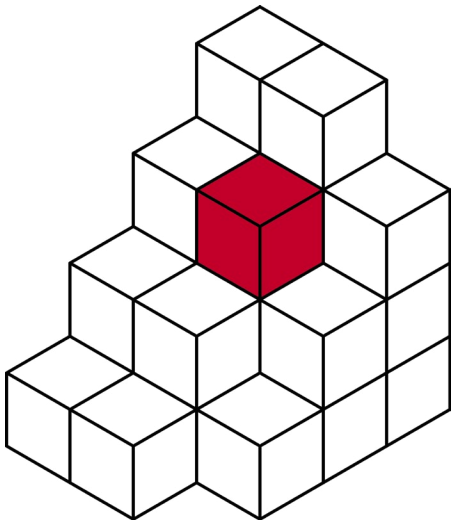
It's easy: we just need to blend together all objects between us and the origin of the ray with their respective alpha values, stacking them with the farthest at the bottom. We initialize a color buffer with alpha to transparent black, then on intersection with an object, we update the buffer by blending the color from the object below our color buffer. Also every step we do this for the dust (we use a r^{-2} density profile). We go on like this until iterations end. Note that the alpha channel then also functions as a z-buffer, as object stop contributing after the ray has intersected an opaque object (that thus set the buffer's alpha to 1.0).

This technique has the obvious drawback that you just cannot stop tracing a ray after you're done with it, as it's part of an array where other rays are still being traced. After colliding with the horizon, for example, rays continue wandering erratically from precision error after they hit the singularity — you can see what happens by explicitly disabling the horizon object. The alpha blending algorithm ensures they won't contribute anymore to the final image — but they will still weigh on the CPU. ■

Riccardo is 23 years old; He is studying for his Master's degree in physics at the University of Padua, under the SGSS. He is aiming in the general direction of high-energy physics and string theory.

Reprinted with permission of the original author.
First appeared in *hn.my/starless* (rantonels.github.io)

The Poor Man's Voxel Engine



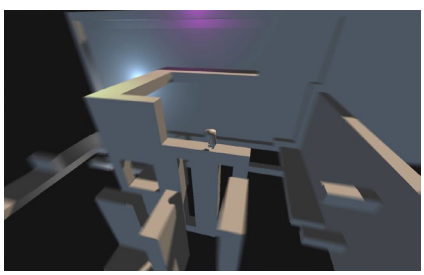
By EVAN TODD

This is not a tutorial. It's a story. A Voxel Odyssey.

The story starts with 19-year-old me in a dorm room next to the Ohio State stadium. I don't have the repo from this stage of development (SVN at the time), but I remember the process clearly.

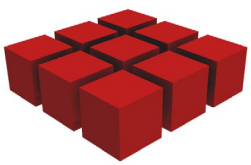
XNA 4 comes out in September, 2010. I immediately dive in. This turns out to be a poor life decision.

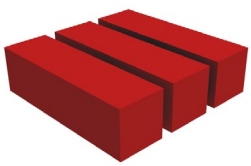
For some reason, one of the very first things I implement is motion blur. I think this is Lemma's first screenshot, although at this point, it's a cartoony third-person game called "Parkour Ninja":

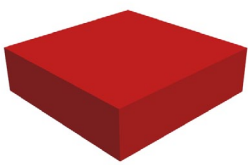


Such motion blur

I skip past the initial naive implementation of spawning a cube for each voxel cell. My first move is to iterate over these individual cells and combine them into larger boxes using run-length encoding.







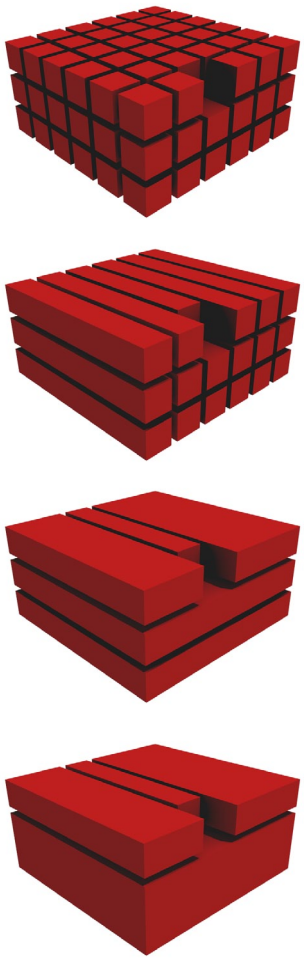
Performance is already a problem even at small scales. I'm re-optimizing the entire scene every time I make an edit. Obviously, my next move is to only optimize the parts I'm editing.

This turns out to be difficult. Take this example:

I add a cube at the top of this stack. To optimize this into a single box, I have to search all the way to the bottom of the stack to find the beginning of the large box, then add 1 to its height and delete my little cube addition.

To speed this process up, I allocate a 3D array representing the entire scene. Each cell stores a pointer to the box it's a part of. Now I can query the cells immediately adjacent to my new addition and quickly get a pointer to the large box next to it.

Removals are the next challenge. My first idea is to split the box back into individual cells, then run the optimizer on them again.



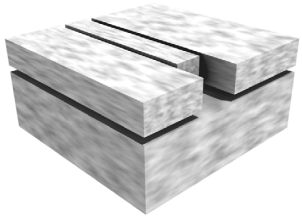
This turns out to be horribly slow. I soon realize that rather than splitting the box into individual cells, I can skip a few steps and split it into "sub-boxes." I still run the optimization algorithm afterward, but I can make its life easier.

Goodbye Xbox

I quickly run into more issues. The CLR's floating point performance is absolutely abysmal on Xbox 360. The physics engine breaks down and cries after spawning 10 boxes or so. I decide to target PCs instead.

Textures

I render scenes by copying, stretching, and scaling a single cube model. Slapping a texture on this cube turns out to be a horrible idea that looks something like this:



To avoid texture stretchiness, I eventually write a shader to generate UVs based on the position and normal of each vertex. Here's the final version for reference:

```
float2x2 UVScaleRotation;
float2 UVOffset;
float2 CalculateUVs(float3 pos,
float3 normal)
{
    float diff = length(pos *
normal) * 2;
    float2 uv = float2(diff +
pos.x + (pos.z * normal.x),
diff - pos.y + (pos.z *
normal.y));
    return mul(uv, UVScaleRota-
tion) + UVOffset;
}
```

Instancing

Next, another performance crisis. Somehow I realize that doing a whole draw call for each and every box in a scene is a Bad Idea. So I take the obvious step and…use hardware instancing. Yes.

Improved level format

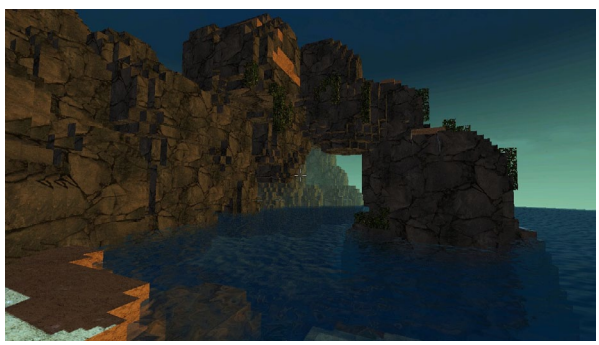
At this point, I'm saving and loading levels via .NET's XML serialization. Apparently XML is still a good idea in 2010. The voxel format is simply a 3D array represented as an XML string of ASCII 1s and 0s. Every time I load a level, I have to re-optimize the entire scene. I solve this by storing the boxes themselves in the level data as a base64 encoded int array. Much better.

Per-surface rendering

I start building larger levels and run into another graphics performance problem. The engine is simply pushing too many triangles. In a complex scene, a significant chunk of boxes are surrounded on all sides by other boxes, completely hidden. But I'm still rendering them.

I solve this problem by breaking each box into its individual faces. I actually iterate across the whole surface to determine what parts (if any) are visible. Shockingly, this turns out to be terrifically slow. I eventually mitigate the issue by caching surface data in the level file.

I render all these surfaces via… drum roll… instancing. Yes, really. I open Blender, create a 1x1 quad, export it, and instance the heck out of that thing. These are dark times. But I'm finally able to render some larger landscapes:



Physics

Time to see some cool physics. I now have two kinds of voxel entities: the static voxel, represented in the physics engine as a series of static boxes, and the dynamic voxel, represented as a single physics entity with a compound collision volume constructed of multiple boxes (I should plug the incredible BEPUPhysics for making this possible). It works surprisingly well. [hn.my/physics]

Around this time I also switch to a deferred renderer, which is why I spawn an unreasonable number of lights at the end of that video.

Chopping down trees

Now it's time to take physics to the next level. My goal is simple: I want the player to be able to cut down a tree and actually see it fall over, unlike Minecraft.

This lofty dream is basically a graph problem, where each box is a node connected to its adjacent neighbors. When I empty out a voxel cell, I need a fast way to determine whether I just partitioned the graph or not.

So I add an adjacency list to the box class. Again, shockingly, calculating adjacency turns out to be a huge bottleneck. I later cache the adjacency data in the level file, which eventually balloons to several megabytes.

Now every time the player empties out a voxel cell, I do a breadth-first search through the entire scene, marking boxes as I go. This allows me to identify "islands" created by the removal of the voxel cell. I can then spawn a new physics entity for that island and break it off from the main scene.

I eventually come up with the idea of "permanent" boxes, which allows me to stop the search whenever I encounter a box that cannot be deleted.

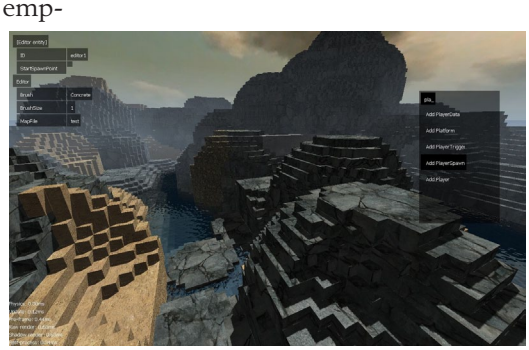
I design a new enemy to showcase the new physics capabilities. I also test the limits of awkwardness and social norms by narrating gameplay footage in a dorm room full of people studying.

Chunks

Around this time I learn about broadphase collision detection. My engine is scattering thousands of static boxes around the world, which makes it difficult for BEPUPhysics' broadphase to eliminate collision candidates. At the same time, it's becoming obvious that rendering the entire world in a single draw call is a bad idea.

I fix both of these issues by splitting the world into chunks. Each chunk has a static triangle mesh for physics, and a vertex buffer with basically the same data for rendering. Since I have to regenerate both of these meshes every time a chunk is modified, the chunk size can't be too large. Also, smaller chunks allow for more accurate frustum culling.

At the same time, the chunks can't be too small, or else the draw call count will explode. After some careful tuning I eventually settle on 80x80x80 chunks.



Partial vertex buffer updates

This is probably the low point.

By now, I am incredibly proud of my "loosely coupled" architecture. I have a Voxel class and a DynamicModel class which know nothing about each other, and a ListBinding between the two which magically transforms a list of Boxes into a vertex buffer.

Somehow, probably through questionable use of the .NET Timer class, I locate a bottleneck: re-sending an entire vertex buffer to the GPU for every voxel mutation is a bad idea. Fortunately, XNA lets me update parts of the vertex buffer individually.

Unfortunately, with all the surface culling I do, I can't tell where to write in the vertex buffer when updating a random box, nor how to shoe-horn this solution into my gorgeous cathedral architecture.

This conundrum occurs during the "dictionary-happy" phase of my career. Yes. The ListBinding now maintains a mapping that indicates the vertex indices allocated for each box. Now I can reach into the vertex buffer and change things without re-sending the whole buffer! And the voxel engine proper still knows nothing about it.

This turned out to never really work reliably.

Multithreading

I lied earlier, this is probably the low point.

Voxel mutations cause noticeable stutters by now. With no performance data to speak of, I decide that multithreading is the answer. Specifically, the worst kind of multithreading.

I spawn a worker thread, sprinkle some locks all over the place, et voilà! It's multithreaded. It gains

perhaps a few milliseconds before the main thread hits an unforeseen mystical code path and the menu somehow manages to acquire a lock on the physics data.

I am ashamed to admit that I never got around to correcting this colossal architectural faux pas.

Large Object Heap

I'm now building large enough levels to run into memory issues. Turns out, the .NET runtime allocates monstrous 80x80x80 arrays differently than your average object.

Long story short, the garbage collector doesn't like to clean up large objects. I end up writing a custom "allocator" that hands out 3D arrays from a common pool. Later, I realize most of the arrays are 90% empty, so I break each chunk into 10x10x10 "sub-chunks" to further reduce memory pressure.

This episode is one of many which explains my present-day distaste for memory-managed languages in game development.

Graduation

I graduate and work at a mobile game studio for the next year. The engine doesn't improve much during this time, but I start to learn that almost everything I know about programming is wrong and incomplete.

I leave my job in February, 2014 and continue hacking the engine. By now it's over 30k LOC and I am morally and spiritually unable to start over on it.

Goodbye allocations

With my newfound awareness of the .NET heap, I realize that my vertex arrays for physics and rendering are also probably landing in the Large Object Heap. Worse, I am reallocating arrays every time they change size, even if only to add a single vertex.

I genericize my Large Object Heap allocator and shove the vertex data in there. Then, rather than allocating arrays at exactly the size I need, I round up to the next power of 2. This cuts the number of allocations and makes it possible for my allocator to reuse arrays more often.

Goodbye cathedral

I finally throw out the "loosely coupled" ListBinding system and pull the vertex generation code into the voxel engine itself. The resulting speed boost is enough for me to go back to resending entire vertex buffers rather than faffing about with partial updates.

Goodbye index buffer

Up to this point, I've been maintaining an index buffer alongside the vertex buffer. In a much overdue stroke of "genius," I realize that since none of the vertices are welded, the index buffer is just a constantly repeating pattern, which is in fact the same for every voxel.

I replace the individual index buffers with a single, global buffer which gets allocated to the nearest power of 2 whenever more indices are needed.

Bit packing and compression

Many numbers in the level data format are guaranteed to fall in the 0-255 range. My friend decides to pack these numbers more efficiently into the integer array.

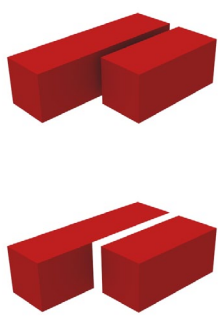
I also pull in third party library #27 (SharpZipLib) and start zipping the level files. These changes cut the file size to under 30% of the original.

Goodbye UV optimization

I've been storing a huge amount of surface data like this:

```
class Box
{
  public struct Surface
  {
    public int MinU, MaxU;
    public int MinV, MaxV;
  }
  public Surface[] Surfaces =
new Surface[]
  {
    new Surface(), // PositiveX
    new Surface(), // NegativeX
    new Surface(), // PositiveY
    new Surface(), // NegativeY
    new Surface(), // PositiveZ
    new Surface(), // NegativeZ
  };
}
```

I do this so that I can resize surfaces that are partially hidden, like this:



At some point in the vertex buffer overhaul, I realize that performance-wise, the physics engine doesn't care what size the surface is.

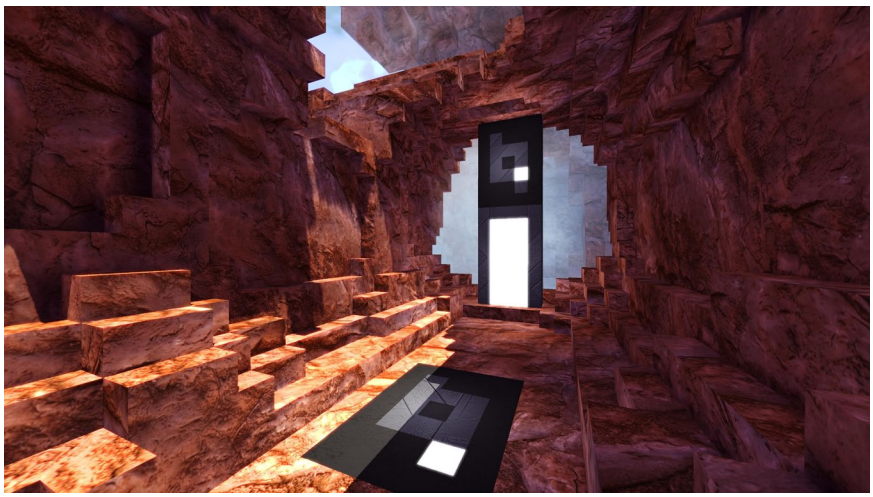
I use this fact to speed up mesh generation. I generate 8 vertices for the corners of each box, then copy them where they need to go in the vertex buffers.

Really, the graphics engine doesn't care much about the size of the surface either, aside from fill rate. What matters is whether the surface is there or not.

With this in mind, I kill the UV optimization code and store the surfaces in memory and in the level file like this:

```
class Box
{
    public int Surfaces;
}
```

The bits of the int are Boolean flags for each surface. Yes, I could do it in a byte. Actually, maybe I should do that. Anyway, this simplifies my level loading and saving code, cuts my file sizes down to about 512kb on average, and drastically reduces memory usage. Axing the UV optimization routine also speeds up mutations.



Conclusion

Clearly, this article is mostly useless if you're interested in writing your own voxel engine. The final result is far from perfect. I just want to share the petty drama of my past four and a half years. I for one thoroughly enjoy reading about other people's struggles. Maybe that's weird.

Lemma [lemmagame.com] is set to release May 2015. The entire game engine is on GitHub. [hn.my/lemma] ■

Evan Todd is a solo independent game developer based in Columbus, Ohio. An avid runner, Vim user, and Pythonista, Evan has developed games since childhood.

Reprinted with permission of the original author.
First appeared in *hn.my/voxel* (et1337.com)

Using Named Pipes and Process Substitution

By VINCE BUFFALO

It's hard not to fall in love with UNIX as a bioinformatician. In a past post [hn.my/xpipe] I mentioned how UNIX pipes are an extremely elegant way to interface bioinformatics programs (and do inter-process communication in general). In exploring other ways of interfacing programs in UNIX, I've discovered two great but overlooked ways of interfacing programs: the named pipe and process substitution.

Why We Love Pipes and UNIX

A few weeks ago I stumbled across a great talk by Gary Bernhardt entitled The UNIX Chainsaw. [hn.my/chainsaw] Bernhardt's "chainsaw" analogy is great: people sometimes fear doing work in UNIX because it's a powerful tool, and it's easy to screw up with powerful tools. I think in the process of grokking UNIX, it's not uncommon to ask "is this clever and elegant? Or completely fucking stupid?" This is normal, especially if you come from a language like Lisp or Python (or even C really). UNIX is a get-shit-done system. I've used a chainsaw, and you're simultaneously amazed at (1) how easily it slices through a tree, and (2) that you're dumb enough to use this thing three feet away from your vital organs. This is UNIX.

Bernhardt also has this great slide, and I'm convinced there's no better way to describe how most UNIX users feel about pipes (especially bioinformaticians):

Pipes are fantastic. Any two (well-written) programs can talk to each other in UNIX. All of the nastiness and the difficulty of inter-process communication is solved with one character, |. Thanks, Doug McIlroy and others. The stream is usually plaintext, the universal interface, but it doesn't have to be. With pipes, it doesn't matter if your pipe is tab delimited marketing data, random email text, or 100 million SNPs. Pipes are a tremendous, beautiful, elegant component of the UNIX chainsaw.

But elegance alone won't earn a software abstraction the hearts of thousands of sysadmins, software engineers, and scientists: pipes are fast. There's little overheard between pipes, and they are certainly a lot more efficient than writing and reading from the disk. In a past article [hn.my/xpipe] I included the classic Samtools pipe for SNP calling. It's no coincidence that other excellent SNP callers like FreeBayes make use of pipes: pipes scale well to moderately large data and they're just plumbing. Interfacing programs this way allows us to check intermediate output for issues, easily rework entire workflows, and even split off a stream with the aptly named program tee.

Where Pipes Don't Work

UNIX pipes are great, but they don't work in all situations. The classic problem is in a situation like this:

```
program --in1 in1.txt --in2 in2.txt --out1 \
  out1.txt --out2 out2.txt > stats.txt 2>  \
  diagnostics.stderr
```

My past colleagues at the UC Davis Bioinformatics Core and I wrote a set of tools for processing next-generation sequencing data and ran into this situation. In keeping with the UNIX traditional, each tool was separate. In practice, this was a crucial design because we saw such differences in data quality due to different sequencing library preparation. Having separate tools working together, in addition to being more UNIX-y, led to more power to spot problems.

However, one step of our workflow has two input files and three output files due to the nature of our data (paired-end sequencing data). Additionally, both `in1.txt` and `in2.txt` were the results of another program, and these could be run in parallel (so interleaving the pairs makes this harder to run in parallel). The classic UNIX pipe wouldn't work, as we had more than one input and output into a file: our pipe abstraction breaks down. Hacky solutions like using standard error are way too unpalatable. What to do?

Named Pipes

One solution to this problem is to use named pipes. A named pipe, also known as a FIFO (after First In First Out, a concept in computer science), is a special sort of file we can create with mkfifo:

```
$ mkfifo fqin
$ prw-r--r-- 1 vinceb  staff 0 Aug  5 22:50 fqin
```

You'll notice that this is indeed a special type of file: p for pipe. You interface with these as if they were files (i.e., with UNIX redirection, not pipes), but they behave like pipes:

```
$ echo "hello, named pipes" > fqin &
[1] 16430
$ cat < fqin
[1]  + 16430 done       echo "hello, named
pipes" > fqin
hello, named pipes
```

Hopefully you can see the power despite the simple example. Even though the syntax is similar to shell redirection to a file, we're not actually writing anything to our disk. Note, too, that the `[1] + 16430` done line is printed because we ran the first line as a background process (to free up a prompt). We could also run the same command in a different terminal window. To remove the named pipe, we just use rm.

Creating and using two named pipes would prevent IO bottlenecks and allow us to interface the program creating `in1.txt` and `in2.txt` directly with program, but I wanted something cleaner. For quick inter-process communication tasks, I really don't want to use `mkfifo` a bunch of times and have to remove each of these named pipes. Luckily UNIX offers an even more elegant way: process substitution.

Process Substitution

Process substitution uses the same mechanism as named pipes, but does so without the need to actually create a lasting named pipe through clever shell syntax. These are also appropriately called "anonymous named pipes." Process substitution is implemented in most modern shells and can be used through the syntax `<(command arg1 arg2)`. The shell runs these commands and passes their output to a file descriptor, which on UNIX systems will be something like `/dev/fd/11`. This file descriptor will then be substituted by your shell where the call to `<()` was. Running a command in parenthesis in a shell invokes a separate subprocess, so multiple uses of `<()` are run in parallel automatically (scheduling is handled by your OS here, so you may want to use this cautiously on shared systems where more explicitly setting the number of processes may be preferable). Additionally, as a subshell, each `<()` can include its own pipes, so crazy stuff like `<(command arg1 | othercommand arg2)` is possible and sometimes wise.

In our simple fake example above, this would look like:

```
program --in1 <(makein raw1.txt) --in2 \
<(makein raw2.txt) --out1 out1.txt --out2 \
out2.txt > stats.txt 2> diagnostics.stderr
```

where makein is some program that creates `in1.txt` and `in2.txt` in the original example (from `raw1.txt` and `raw2.txt`) and outputs it to standard out. It's that simple: you're running a process in a subshell, and its standard out is going to a file descriptor (the `/dev/fd/11` or whatever number it is on your system), and program is taking input from that. In fact, if we see this process in `htop` or with `ps`, it looks like:

```
$ ps aux | grep program
vince  [...] program --in1 /dev/fd/63 --in2 /
dev/fd/62 --out1 out1.txt --out2 out2.txt >
stats.txt 2> diagnostics.stderr
```

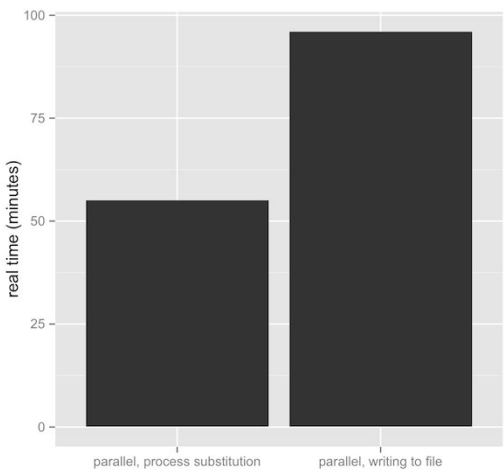
But suppose you wanted to pass `out1.txt` and `out2.txt` to gzip to compress them? Clearly we don't want to write them to disk and then compress them, as this is slow and a waste of system resources. Luckily process substitution works the other way, too, through `>()`. So we could compress in place with:

```
program --in1 <(makein raw1.txt) --in2 \
<(makein raw2.txt) --out1 >(gzip > \
out.txt.gz) --out2 >(gzip > out2.txt.gz) \
> stats.txt 2> diagnostics.stderr
```

UNIX never ceases to amaze me in its power. The chainsaw is out and you're cutting through a giant tree. But power comes with a cost here: clarity. Debugging this can be difficult. This level of complexity is like Marmite: I recommend not layering it on too thick at first. You'll hate it and want to vomit. Admittedly, the nested inter-process communication syntax is neat but awkward — it's not the simple, clearly understandable | that we're used to.

Speed

So, is this really faster? Yes, quite. Writing and reading to the disk comes at a big price — see latency numbers every programmer should know. [hn.my/latency] Unfortunately I am too busy to do extensive benchmarks, but I wrote a slightly insane read trimming script [hn.my/trim] that makes use of process substitution. Use at your own risk, but we're using it over simple Sickle/Scythe/Seqqs combinations. One test uses `trim.sh`, the other is a simple shell script that just runs Scythe in the background (in parallel, combined with Bash's wait), writes files to disk, and Sickle processes these. The benchmark is biased against process substitution, because I also compress the files via `>(gzip > )` in those tests, but don't compress the others. Despite my conservative benchmark, the difference is striking:



Additionally, with the `>(gzip > )` bit, our sequences had a compression ratio of about 3.46% — not bad. With most good tools handling gzip compression natively (that is, without requiring prior decompression), and easy in-place compression via process substitution, there's really no reason to not keep large data sets compressed. This is especially the case in bioinformatics where we get decent compression ratios, and our friends `less`, `cat`, and `grep` have their `zless`, `gzcat`, and `zgrep` analogs.

Once again, I'm astonished at the beauty and power of UNIX. As far as I know, process substitution is not well known — I asked a few sysadmin friends, and they'd seen named pipes but not process substitution. But given UNIX's abstraction of files, it's no surprise. Actually Brian Kernighan waxed poetically about both pipes and UNIX files in this classic AT&T 1980s video on UNIX. [hn.my/1980] Hopefully younger generations of programmers will continue to discover the beauty of UNIX (and stop re-inventing the wheel, something we've all been guilty of). Tools that are designed to work in the UNIX environment can leverage UNIX's power and end up with emergent powers. ■

Vince Buffalo is a graduate student in the Population Biology Graduate Group at UC Davis studying evolutionary genetics and statistics. Before starting his PhD, Vince worked as a bioinformatician at the UC Davis Genome Center and Department of Plant Sciences. Vince is author of the O'Reilly book Bioinformatics Data Skills to be published in May 2015.

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

Dad and the Ten Commandments of Egoless Programming

By STEPHEN WYATT BUSH

Dad and I got to talk about programming for two weeks before he died.

I was 22, a senior in college completing a BFA in graphic design. Dad was 62, an older dad than most. When he started programming at Tennessee Tech back in the '60s, he wrote FORTRAN on punch cards. He was a wealth of knowledge.

I had just been introduced to code that semester, and it was already consuming my thoughts. It felt magical and powerful, in many ways a more fulfilling creative practice than visual design (but that's for another post).

When I came home for the holidays, Dad shared The Ten Commandments of Egoless Programming with me. He printed them and we discussed each point. It was one of the few programming related things we were able to discuss before he unexpectedly passed; perhaps that's why it sticks with me.

From The Psychology of Computer Programming, written in 1971 by Jerry Weinberg, here are The Ten Commandments of Egoless Programming:

1. **Understand and accept that you will make mistakes.** The point is to find them early, before they make it into production. Fortunately, except for the few of us developing rocket guidance software at JPL, mistakes are rarely fatal in our industry. We can, and should, learn, laugh, and move on.

2. **You are not your code.** Remember that the entire point of a review is to find problems, and problems will be found. Don't take it personally when one is uncovered.

3. **No matter how much "karate" you know, someone else will always know more.** Such an individual can teach you some new moves if you ask. Seek and accept input from others, especially when you think it's not needed.

4. **Don't rewrite code without consultation.** There's a fine line between "fixing code" and "rewriting code." Know the difference, and pursue stylistic changes within the framework of a code review, not as a lone enforcer.

5. **Treat people who know less than you with respect, deference, and patience.** Non-technical people who deal with developers on a regular basis almost universally hold the opinion that we are prima donnas at best and crybabies at worst. Don't reinforce this stereotype with anger and impatience.

> “The only true authority stems from knowledge, not from position.”

6 **The only constant in the world is change.** Be open to it and accept it with a smile. Look at each change to your requirements, platform, or tool as a new challenge, rather than some serious inconvenience to be fought.

7 **The only true authority stems from knowledge, not from position.** Knowledge engenders authority, and authority engenders respect — so if you want respect in an egoless environment, cultivate knowledge.

8 **Fight for what you believe, but gracefully accept defeat.** Understand that sometimes your ideas will be overruled. Even if you are right, don’t take revenge or say “I told you so.” Never make your dearly departed idea a martyr or rallying cry.

9 **Don’t be “the coder in the corner.”** Don’t be the person in the dark office emerging only for soda. The coder in the corner is out of sight, out of touch, and out of control. This person has no voice in an open, collaborative environment. Get involved in conversations, and be a participant in your office community.

10 **Critique code instead of people.** Be kind to the coder, not to the code. As much as possible, make all of your comments positive and oriented to improving the code. Relate comments to local standards, program specs, increased performance, etc.

I keep this list around even today. It has already helped me be a better programmer. Sometimes I imagine what other bits of advice he’d give me were he still around. While I cannot know, I feel sure he’d be proud so long as I keep these in mind.

For more on Dad, read Frank Bush’s Contributions to the Computing Profession, compiled by his coworkers at TTU. [hn.my/frankbush] ■

Stephen Wyatt Bush is an artist and software engineer living in San Francisco. He works at Airbnb.

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

DevOps Is Bullshit

Why One Programmer Doesn't Do It Anymore

By JOSH JOHNSON

I've always been handy with hardware. I was one of "those kids" you hear about that keeps taking things apart just to see how they work — and driving their parents nuts in the process. When I was a teenager, I toyed with programming but didn't get serious with it until I decided I wanted to get into graphic design. I found out that you don't have to write HTML yourself, you can use programming to do it for you!

But I never stopped tinkering with hardware and systems. I used Linux and BSD on my desktop for years, built my LAMP stacks from source, and simulated the server environment when I couldn't — when I used Windows for work, and when I eventually adopted Apple as my primary platform, I first started with cross-compiled versions of the components, and eventually got into virtualization.

In the early days (maybe 10 years ago), there seemed to be few programmers who were like me, or if they were, they never took "operations" or "sysadmin" jobs, and neither did I. So there was always a natural divide. Aside from being a really nice guy who everyone likes, I had a particular rapport with my cohorts who specialized in systems.

I'm not sure exactly what it was. It may have been that I was always interested in the finer details of how a system works. It may have been my tendency to document things meticulously, or my interest in automation and risk reduction. It could have just been that I was willing to take the time to cross the divide and talk to them, even when I didn't need something. It may have just boiled down to the fact that when they were busy, I could do things myself, and I wanted to follow their standards and get their guidance. It's hard to tell, even today, as my systems skills have developed beyond what they ever were before, but the rapport has continued on.

And then something happened. As my career progressed, I took on more responsibilities and did more and more systems work. This was partly because of the divide widening to some extent at one particular job, but mostly because, I could. Right around this time the "DevOps Revolution" was beginning.

Much like when I was a teenager and everyone needed a web site, suddenly everyone needed DevOps.

I didn't really know what it was. I was aware of the term, but being a smart person, I tend to ignore radical claims of great cultural shifts, especially in technology. In this stance, I find myself feeling a step or two behind at times, but it helps keep things in perspective. Over time, technology changes, but true radicalism is rare. Most often, a reinvention or revisiting of past ideas forms the basis for such claims. This "DevOps" thing was no different. Honestly, at the time it seemed like a smoke screen; a flashy way to save money for startups.

I got sick of tending systems — when you're doing it properly, it can be a daunting task. Dealing with storage, access control, backups, networking, high availability, maintenance, security, and all of the domain-specific aspects can easily become overwhelming. But worse, I was doing too much front-line

support, which honestly, at the time was more important than the programming it was distracting me from. I love my users, and I see their success as my success. I didn't mind it, but the bigger problems I wanted to solve were consistently being held above my head, just out of my grasp. I could ignore my users or ignore my passion, and that was a saddening conundrum. I felt like all of the creativity I craved was gone, and I was being paid too much (or too little depending on if you think I was an over paid junior sysadmin or an under paid IT manager with no authority) to work under such tedium. So I changed jobs.

I made the mistake of letting my new employer decide where they wanted me to go in the engineering organization.

What I didn't know about this new company was that it had been under some cultural transition just prior to bringing me on board. Part of that culture shift was incorporating so-called "DevOps" into the mix. By fiat or force.

Because of my systems experience, I landed on the front line of that fight: the "DevOps Team." I wasn't happy.

But as I dug in, I saw some potential. We had the chance to really shore up the development practices, reduce risk in deployments, make the company more agile, and ultimately make more money.

We had edicts to make things happen, under the assumption that if we built it, the developers would embrace it. These things included continuous integration, migrating from subversion to git, building and maintaining code review tools, and maintaining the issue tracking system. We had other, less explicit responsibilities that became central to our work later on, including developer support, release management, and interfacing with the separate, segregated infrastructure department. This interaction was especially important, since we had no systems of our own, and we weren't allowed to administer any machines. We didn't have privileged access to any of the systems we needed to maintain for a long time.

With all the hand wringing and flashing of this "DevOps" term, I dug in and read about it and what all the hubbub was about. I then realized something. What we were doing wasn't DevOps.

Then I realized something else. I was DevOps. I always had been. The culture was baked into the kind of developer I was. Putting me and other devs with similar culture on a separate team, whether that was the "DevOps" team or the infrastructure team was a fundamental mistake.

The developers didn't come around. At one point someone told a teammate of mine that they thought we were "IT support." What needed to happen was the developers had to embrace the concept that they were capable of doing at least some systems things themselves, in safe and secure manner, and the infrastructure team had to let them do it. But my team just sat there in the middle, doing what we could to keep the lights on and get the code out, but ultimately just wasting our time. Some developers starting using AWS, with the promise of it being a temporary solution, but in a vacuum nonetheless. We were not having the impact that management wanted us to have.

My time at this particular company ended in a coup of sorts. This story is worthy of a separate blog post some day when it hurts a little less to think about. But let's just say I was on the wrong side of the revolution and left as quickly as I could when it was over.

In my haste, I took another "DevOps" job. My manager there assured me that it would be a programming job first and a systems job second. "We need more "dev" in our "devops,"" he told me.

What happened was very similar to my previous "DevOps" experience, but more acute. Code, and often requirements, were thrown over the wall at the last minute. As it fell in our laps, we scrambled to make it work properly, as it seemed no one would think of things like fail over or backups or protecting private information when they were making their plans. Plans made long ago, far away, and without our help.

This particular team was more automation focused. We had two people who were more "dev" than "ops," and the operations people were no slouches when it came to scripting or coding in their own right.

It was a perfect blend, and as a team we got along great and pulled off some miracles.

But ultimately, we were still isolated. We and our managers tried to bridge the gap, to no avail. Developers, frustrated with our sizable backlog, went over our heads to get access to our infrastructure and started doing it for themselves, often with little or no regard for our policies or practice. We would be tasked with cleaning up their mess when it was time for production deployment — typically in a major hurry after the deadline had passed.

> “At the companies I interviewed at, it seemed that “DevOps” really meant “operations and automation”, or more literally “AWS and configuration management”.”

The original team eventually evaporated. I was one of the last to leave, as new folks were brought into a remote office. I stuck it out for a lot of reasons: I was promised transfer to NYC, I had good healthcare, and I loved my team. But ultimately what made me stick around was the hope that kept getting rebuilt and dashed as management rotated in and out above me: that we could make it work.

I took the avenue of providing automated tools to let the developers have freedom to do as they pleased, yet we could ensure they were complying with company security guidelines and adhering to sane operations practices.

Sadly, politics and priorities kept my vision from coming to reality. It’s OK, in hindsight, because so much more was broken about so-called “DevOps” at this particular company. I honestly don’t think that it could have made that much of a difference.

Near the end of my tenure there, I tried to help some of the developers help themselves by sitting with them and working out how to deploy their code properly side-by-side. It was a great collaboration, but it fell short. It represented a tiny fraction of the developers we supported. Even with those really great developers finally interfacing with my team, it was too little, too late.

Another lesson learned: you can’t force cultural change. It has to start from the bottom up, and it needs breathing room to grow.

I had one final “DevOps” experience before I put my foot down and made the personal declaration that “DevOps is bullshit,” and I wasn’t going to do it anymore.

Due to the titles I had taken, and the experiences of the last couple of years, I found myself in a predicament. I was seen by recruiters as a “DevOps guy” and not as a programmer. It didn’t matter that I had 15 years of programming experience in several languages, or that I had focused on programming even in these so-called “DevOps” jobs. All that mattered was that, as a “DevOps Engineer” I could be easily packaged for a high-demand market.

I went along with the type casting for a couple of reasons. First, as I came to realize, I am DevOps — if anyone was going to come into a company and bridge the gap between operations and engineering, it’d be me. Even if the company had a divide, which every company I interviewed with had, I might be able to come on board and change things.

But there was a problem. At least at the companies I interviewed at, it seemed that “DevOps” really meant “operations and automation” (or more literally “AWS and configuration management”). The effect this had was devastating. The somewhat superficial nature of parts of my systems experience got in the way of landing some jobs I would have been great at. I was asked questions about things that had never been a problem for me in 15 years of building software and systems to support it, and being unable to answer, but happy to talk through the problem, would always end in a net loss.

> **Apply the practices and technology that comprise what DevOps is to your development process, and stop putting up walls between different specialties.**

When I would interview at the few programming jobs I could find or the recruiters would give me, they were never for languages I knew well. And even when they were, my lack of computer science jargon bit me — hard. I am an extremely capable software engineer, someone who learns quickly and hones skills with great agility. My expertise is practical, however, and it seemed that the questions that needed to be asked, that would have illustrated my skill, weren't. I think to them, I looked like a guy who was sick of systems that was playing up their past dabbling in software to change careers.

So it seemed "DevOps," this great revolution, and something that was baked into my very identity as a programmer, had left me in the dust.

I took one final "DevOps" job before I gave up. I was optimistic, since the company was growing fast and I liked everyone I met there. Sadly, it had the same separations and was subject to the same problems. The developers, whom I deeply respected, were doing their own thing in a vacuum. My team was unnecessarily complicating everything and wasting huge amounts of time. Again, it was just "ops with automation" and nothing more.

So now let's get to the point of all of this. We understand why I might think "DevOps is bullshit," and why I might not want to do it anymore. But what does that really mean? How can my experiences help you, as a developer, as an operations person, or as a company with issues they feel "DevOps" could address?

Don't *do* DevOps. It's that simple. Apply the practices and technology that comprise what DevOps is to your development process, and stop putting up walls between different specialties.

A very wise man once said "If you have a DevOps team, you're doing it wrong." If you start doing that, **stop it**.

There is some nuance here, and my experience can help save you some trouble by identifying some of the common mistakes:

- DevOps doesn't make specialists obsolete.
- Developers can learn systems and operations, but nothing beats experience.
- Operations people can learn development, too, but again, nothing beats experience.
- Operations and development have historically be separated for a reason — there are compromises you must make if you integrate the two.
- Tools and automation are not enough.
- Developers have to want DevOps. Operations have to want DevOps. **At the same time.**
- Using "DevOps" to save money by reducing staff will blow up in your face.
- You can't have DevOps and still have separate operations and development teams. **Period.**

Let me stop for one moment and share another lesson I've learned: if it ain't broke, don't fix it.

If you have a working organization that seems old fashioned, **leave it alone.** It's possible to incorporate the tech, and even some of the cultural aspects of DevOps without radically changing how things work — it's just not DevOps anymore, so don't call it that. Be critical of your

process and practices, kaizen and all that, but don't sacrifice what works just to join the cargo cult. You will waste money, and you will destroy morale. The pragmatic operations approach is the happiest one.

Beware of geeks bearing gifts.

So let's say you know why you want DevOps, and you're certain that the cultural shift is what's right for your organization. Everyone is excited about it. What might a proper "DevOps" team look like?

I can speak to this, because I currently work in one.

First, never call it "DevOps." It's just what you do as part of your job. Some days you're writing code, other days you're doing a deployment or maintenance. Everyone shares all of those responsibilities equally.

People still have areas of experience and expertise. This isn't pushing people into a lukewarm, mediocre dilution of their skills — this is passionate people doing what they love. It's just that part of that is launching a server or writing a chef recipe or debugging a production issue.

As such you get a truly cross-functional team. Where expertise differs, there's a level of respect and trust. So if someone knows more about a topic than someone else, they will likely be the authority on it. The rest of the team trusts them to steer the group in the right direction.

This means that you can hire operations people to join your team. Just don't give them exclusive responsibility for what they're best at — integrate them. The same goes for any "non developer" skillset, be that design, project management, or whatever.

Beyond that, everyone on the team has a thirst to develop new skills and look at their work in different ways. This is when the difference in expertise provides an opportunity to teach. Teaching brings us closer together and helps us all gain better understanding of what we're doing.

So that's what DevOps really is. You take a bunch of really skilled, passionate, talented people who don't have their heads shoved so far up their own asses that they can take the time to learn new things. People who see the success of the business as a combined responsibility that is equally shared. "That's not my job" is not something they are prone to saying, but they're happy to delegate or share a task if need be. You give them the infrastructure, and time (and encouragement doesn't hurt), to build things in a way that makes the most sense for their productivity and the business, embracing that equal, shared sense of responsibility. Things like continuous integration and zero-downtime deployments just happen as a function of smart, passionate people working toward a shared goal.

It's an organic, culture-driven process. We may start doing continuous deployment, or utilize "the cloud" or treat our "code as infrastructure," but only if it makes sense. The developers are the operations people and the operations people are the developers. An application system is seen in a holistic manner and developed as a single unit. No one is compromising, we all get better as we all just fucking do it.

DevOps is indeed bullshit. What matters is good people working together without artificial boundaries. Tech is tech. It's not possible for everyone to share like this, but when it works, it's amazing — but is it really DevOps? I don't know, I don't do that anymore. ■

Josh Johnson is a engineer of allthethings, with a passion for problem solving. He dislikes magic, and prefers unicorns to roam free.

Reprinted with permission of the original author. First appeared in *hn.my/devops* (lionfacelemonface.wordpress.com)

Embracing SQL In Postgres

By ROB CONERY

One thing that drives me absolutely over the cliff is how ORMs try so hard (and fail) to abstract the power and expressiveness of SQL. Before I write further let me say that Frans Bouma reminded me yesterday there's a difference between ORMs and the people that use them. They're just tools (the ORMs) — and I agree with that in the same way I agree that crappy fast food doesn't make people fat — it's the people that eat too much of it.

Instead of ripping ORMs apart again — I'd like to be positive and tell you just why I have stopped using their whack-ass OO abstraction on top of my databases. In short: *it's because SQL can expertly help you express the value of your application in terms of the data*. That's really the only way you're going to know whether your app is any good: by the data it generates.

So give it a little of your time — it's fun once you get rolling with the basics and how your favorite DB engine accentuates the SQL standard. Let's see some examples. (By the way all of what I'm using below is detailed here in the Postgres docs. [hn.my/psqldocs] Have a read, there's a lot of stuff you can learn. My examples below barely even scratch the surface.)

Postgres Built-in Fun

Right from the start: *Postgres sugary SQL syntax is really, really fun*. SQL is an ANSI standardized language — this means you can roughly expect to have the same rules from one system to the next (which means you can't expect it at all).

Postgres follows the standards almost to the letter — but it goes beyond with some very fun additions. Let's take a look!

Regex

At some point you might need to run a rather complicated string matching algorithm. Many databases (including SQL Server) allow you to use Regex patterning through a function or some other construct. With Postgres it works in a lovely, simple way (using PSQL for this with the old Tekpub database):

```
select sku,title from products where title ~*
'master';
    sku     |              title
------------+---------------------------------
 aspnet4    | Mastering ASP.NET 4.0
 wp7        | Mastering Windows Phone 7
 hg         | Mastering Mercurial
 linq       | Mastering Linq
 git        | Mastering Git
 ef         | Mastering Entity Framework 4.0
 ag         | Mastering Silverlight 4.0
 jquery     | Mastering jQuery
 csharp4    | Mastering C# 4.0 with Jon Skeet
 nhibernate | Mastering NHibernate 2
(10 rows)
```

The `~*` operator says "here comes a POSIX regex pattern that's case insensitive." You can make it case sensitive by omitting the `*`.

Regex can be a pain to work with but if you wanted to, you could ramp this query up by using Postgres' built-in Full Text indexing:

```
select products.sku,
products.title
from products
where to_tsvector(title) @@ to_tsquery('Mastering');
```

```
    sku     |              title
------------+---------------------------------
 aspnet4    | Mastering ASP.NET 4.0
 wp7        | Mastering Windows Phone 7
 hg         | Mastering Mercurial
 linq       | Mastering Linq
 git        | Mastering Git
 ef         | Mastering Entity Framework 4.0
 ag         | Mastering Silverlight 4.0
 jquery     | Mastering jQuery
 csharp4    | Mastering C# 4.0 with Jon Skeet
 nhibernate | Mastering NHibernate 2
(10 rows)
```

This is a bit more complicated. Postgres has a built-in data type specifically for the use of Full Text indexing — `tsvector`. You can even have this as a column on a table if you like, which is great as it's not hidden away in some binary index somewhere.

I'm converting my title on the fly to `tsvector` using the `to_tsvector()` function. This tokenizes and prepares the string for searching. I'm then shoving this into the `to_tsquery()` function. This is a query built from the term "Mastering". The `@@` bits simply say "return true if the `tsvector` field matches the `tsquery`". The syntax is a bit wonky, but it works really well and is quite fast.

You can use the `concat` function to push strings together for use on additional fields, too:

```
select products.sku,
products.title
from products
where to_tsvector(concat(title,' ',description))
@@ to_tsquery('Mastering');
    sku     |              title
------------+---------------------------------
 aspnet4    | Mastering ASP.NET 4.0
 wp7        | Mastering Windows Phone 7
 hg         | Mastering Mercurial
 linq       | Mastering Linq
 git        | Mastering Git
 ef         | Mastering Entity Framework 4.0
 ag         | Mastering Silverlight 4.0
 jquery     | Mastering jQuery
 csharp4    | Mastering C# 4.0 with Jon Skeet
 nhibernate | Mastering NHibernate 2
(10 rows)
```

This combines `title` and `description` into one field and allows you to search them both at the same time using the power of a kick-ass full text search engine. I could spend multiple posts on this — for now just know you can do it inline.

Generating a Series

One really fun function that's built in is `generate_series()` — it outputs a sequence that you can use in your queries for any reason:

```
select * from generate_series(1,10);
 generate_series
-----------------
               1
               2
               3
               4
               5
               6
               7
               8
               9
              10
```

If sequential things aren't what you want, you can order by another great function — `random()`:

```
select * from generate_series(1,10,2)
order by random();
 generate_series
-----------------
               3
               5
               7
               1
               9
(5 rows)
```

Here I've added an additional argument to tell it to skip by 2.

It also works with dates:

```
select * from generate_series(
         '2014-01-01'::timestamp,
         '2014-12-01'::timestamp,
         '42 days');

   generate_series
---------------------
 2014-01-01 00:00:00
 2014-02-12 00:00:00
 2014-03-26 00:00:00
 2014-05-07 00:00:00
 2014-06-18 00:00:00
 2014-07-30 00:00:00
 2014-09-10 00:00:00
 2014-10-22 00:00:00
(8 rows)
```

Here I'm telling it to output the dates in 2014 in 42 day intervals. You can do this backwards, too; you just have to use a negative interval.

Why is this useful? You can alias this function and plug in the number from the series generation into whatever calculation you want:

```
select x as first_of_the_month from
generate_series('2014-01-01'::timestamp,'2014-
12-01'::timestamp,'1 month') as f(x);
first_of_the_month
---------------------
 2014-01-01 00:00:00
 2014-02-01 00:00:00
 2014-03-01 00:00:00
 2014-04-01 00:00:00
 2014-05-01 00:00:00
 2014-06-01 00:00:00
 2014-07-01 00:00:00
 2014-08-01 00:00:00
 2014-09-01 00:00:00
 2014-10-01 00:00:00
 2014-11-01 00:00:00
 2014-12-01 00:00:00
(12 rows)
```

Aliasing functions like this allows you to use the resulting row inline with your SQL call. This kind of thing is nice for analytics and spot-checks on your data. Also, notice the `month` specification? That's an interval in Postgres — something you'll use a lot with data stuff. Speaking of dates…

Date Math Fun

Intervals are brilliant shortcuts for working with dates in Postgres. For instance, if you want to know the date 1 week from today…

```
select '1 week' + now() as a_week_from_now;
        a_week_from_now
-------------------------------
 2015-03-03 10:08:12.156656+01
(1 row)
```

Postgres sees `now()` as a `timestamp` and uses the `+` operator to infer the string "1 week" as an interval. Brilliant. But do you notice the result `2015-03-03 10:08:12.156656+01`? This is a very interesting thing!

It's telling me the current date and time all the way down to milliseconds… and also the timezone (+1 as I'm currently in Italy).

If you've ever had to wrestle with dates and UTC — well, it's a major pain. Postgres has a built-in `timestamptz` data type — timestamp with time zone — that will account for this when doing date calculations.

This is really fun to play with. For instance I can ask Postgres what time it is in California:

```
SELECT now() AT TIME ZONE 'PDT' as cali_time;
         cali_time
----------------------------
 2015-02-24 02:16:57.884518
(1 row)
```

2am — best not to call Jon Galloway and tell him his SQL Server is on fire. This returns an interval — the difference between two timestamps.

How many hours behind me is Jon? Let's see…

```
select now() - now() at time zone 'PDT' as cali_
diff;
 cali_diff
-----------
 08:00:00
(1 row)
```

Notice the return value is a `timestamp` of 8 hours, not an integer. Why is this important? Time is a relative thing and it's incredibly important to know which time zone your server is in when you calculate things based on time.

For instance — in my Tekpub database I recorded when orders were placed. If 20 orders came in during that "End of the Year Sale," my accountant would very much like to know if they came in before, or after,

midnight on January 1, 2013. My server is in New York, but my business is registered in Hawaii.

This is important stuff, and Postgres handles this and many other date functions quite nicely.

Aggregates

Working with rollups and aggregates in Postgres can be tedious precisely because it's so very, very standards-compliant. This always leads to having to be sure that whatever you GROUP BY is in your `SELECT` clause.

Meaning, if I want to look at sales for the month, grouped by week I'd need to run a query like this:

```
select sku, sum(price),
date_part('month',created_at) from invoice_items
group by sku,date_part('month',created_at)
having date_part('month',created_at) = 9
```

That's a bit extreme and a bit of a PITA to write (and remember the syntax!). Let's use a better SQL feature in Postgres: windowing functions:

```
select distinct sku, sum(price) OVER (PARTITION
BY sku)
from invoice_items
where date_part('month',created_at) = 9
```

Same data, less noise (windowing functions are also available in SQL Server). Here I'm doing set-based calculations by specifying I want to run a `SUM` over a partition of data for a given row. If I didn't specify `DISTINCT` here, the query would have spit out all sales as if it we just a normal `SELECT` query.

The nice thing about using windowing functions is that I can pair aggregates together:

```
select distinct sku, sum(price) OVER (PARTITION
BY sku) as revenue,
count(1) OVER (PARTITION BY sku) as sales_count
from invoice_items
where date_part('month',created_at) = 9
```

This gives me a monthly sales count per sku as well as revenue. I can also output total sales for the month in the very next column:

```
select distinct sku,
sum(price) OVER (PARTITION BY sku) as revenue,
count(1) OVER (PARTITION BY sku) as sales_count,
sum(price) OVER (PARTITION by 0) as sales_total
from invoice_items
where date_part('month',created_at) = 9
```

I'm using `PARTITION BY 0` here as a way of saying "just use the entire set as the partition" — this will rollup all sales for September.

Combine this with the power of a Common Table Expression, and I can run some interesting calcs:

```
with september_sales as (
    select distinct sku,
    sum(price) OVER (PARTITION BY sku) as rev-
enue,
    count(1) OVER (PARTITION BY sku) as sales_
count,
    sum(price) OVER (PARTITION by 0) as sales_
total
    from invoice_items
    where date_part('month',created_at) = 9
)

select sku,
    revenue::money,
    sales_count,
    sales_total::money,
    trunc((revenue/sales_total * 100),4) as per-
centage
from september_sales
```

In the final select I'm casting `revenue` and `sales_total` as `money` — which means it will be formatted nicely with a currency symbol.

A pretty comprehensive sales query — I get a total per sku, a sales count and a percentage of monthly sales with (what I promise becomes) fairly straightforward SQL.

I'm using `trunc` in the CTE here to round to 4 significant digits as the percentages can be quite long.

Strings

I showed you some fun with Regex above, but there is more you can do with strings in Postgres. Consider this query, which I used quite often (again, the Tekpub database):

```
select products.sku,
    products.title,
    downloads.list_order,
    downloads.title  as episode
from products
inner join downloads on downloads.product_id =
products.id
order by products.sku, downloads.list_order;
```

This fetched all of my videos and their individual episodes (I called them downloads). I would use this query on display pages, which worked fine.

But what if I just wanted an episode summary? I could use some aggregate functions to this. The simplest first — just a comma-separated string of titles:

```
select products.sku,
    products.title,
    string_agg(downloads.title, ', ') as down-
loads
from products
inner join downloads on downloads.product_id =
products.id
group by products.sku, products.title
order by products.sku
```

`string_agg` works like `String.join()` in your favorite language. But we can do one better: let's concatenate and send things down in an array for the client:

```
select products.sku,
    products.title,
    array_agg(concat(downloads.list_order,')
',downloads.title)) as downloads
from products
inner join downloads on downloads.product_id =
products.id
group by products.sku, products.title
order by products.sku
```

Here I'm using `array_agg` to pull in the `list_order` and `title` from the joined downloads table and output them inline as an array. I'm using the `concat` function to concatenate a pretty title using the `list_order` as well.

If you're using Node, this will come back to you as an array you can iterate over.

If you're using Node, you'll probably want to have this JSON'd out, however:

```
select products.sku,
    products.title,
    json_agg(downloads) as downloads
from products
inner join downloads on downloads.product_id =
products.id
group by products.sku, products.title
order by products.sku
```

Here I'm shoving the related downloads bits (aka the "Child" records) into a field that I can easily consume on the client — an array of JSON.

Summary

If you don't know SQL very well — particularly how your favorite database engine implements and enhances it — take this week to get to know it better. It's so very powerful for working the gold of your application: your data. ■

Rob Conery co-founded Tekpub. He used to work at Microsoft on the ASP.NET team and have led a number of Open Source projects in the Microsoft realm.

Reprinted with permission of the original author.
First appeared in *hn.my/sql* (conery.io)

The $3500 Shirt

A History Lesson in Economics

By EVE FISHER

One of the great advantages of being a historian is that you don't get your knickers in as much of a twist over how bad things are today. If you think this year is bad, try 1347, when the Black Death covered most of Europe, one third of the world had died, and (to add insult to injury) there was also (in Europe) the little matter of the Hundred Years' War and the Babylonian Captivity of the Church (where the pope had moved to Avignon, France, and basically the Church was being transformed into a subsidiary of the French regime). Things are looking up already, aren't they?

Another thing is economics. Everyone complains about taxes, prices, and how expensive it is to live anymore. I'm not going to go into taxes — that way lies madness. But I can tell you that living has never been cheaper. We live in a country awash in stuff — food, clothing, appliances, machines, cheap crap from China — but it's never enough. $4 t-shirts? Please. We want five for $10, and even then, can we get them on sale? And yet, compared to a world where everything is made by hand — we're talking barely 200 years ag— everything is cheap and plentiful, and we are appallingly ungrateful.

Let's talk clothing. When the Industrial Revolution began, it started with factories making cloth. Why? Because clothing used to be frighteningly expensive. Back in my teaching days I gave a standard lecture, which is about to follow, on the $3,500 shirt, or why peasants owned so little clothing. Here's the way it worked:

See this guy below, front left dancing? He's wearing a standard medieval shirt. It has a yoke, a bit of smocking and gathering around the neck, armholes, and the wrists would be banded, so he could tie or button them closed.

Oh, and in the Middle Ages, it would be expected that all of the inside sleeves would be finished. This was all done by hand. A practiced seamstress could probably sew it in 7 hours. But that's not all that would go into the making. There's the cloth. A shirt like this would take about 4 yards of cloth, and it would be a fine weave: the Knoxville Museum of Art estimates 2 inches an hour. So 4(yards)*36(inches)/2 = 72 hours. (I'm a weaver — or at least I used to be — so this sounds accurate to me.) Okay, so hand weaving and hand sewing would take 79 hours. Now the estimate for spinning has always been complex, so stick with me for a minute: Yardage of thread for 4 yards of cloth, 1 yard wide (although old looms often only wove about 24" wide cloth), and requires 12 threads per inch, so:

12 threads * 36“ wide * (4 yards + 2 yards for tie-up = 6 yards, or 72”) * 72 = 31,004 inches, or 864 yards of thread for the warp. And you'd need about the same for a weft, or a total of about 1600 yards of thread for one shirt.

1600 yards would take a while to spin. At a Dark Ages recreation site, they figured out a good spinner could do 4 yards in an hour, so that would be 400 hours to make the thread for the weaving.

So, 7 hours for sewing, 72 for weaving, 400 for spinning, or 479 hours total to make one shirt. At minimum wage ($7.25/hour) that shirt would cost $3,472.75.

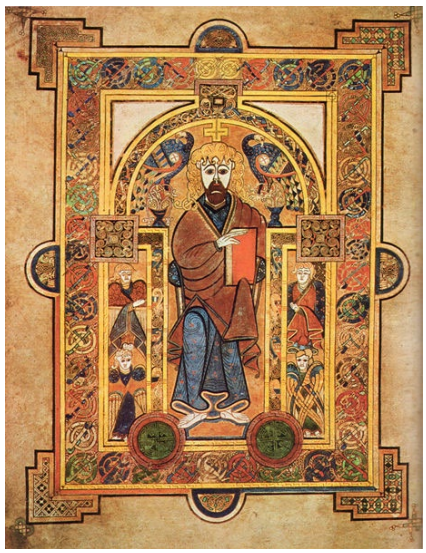
And that's just a standard shirt.

And that's not counting the work that goes into raising sheep or growing cotton and then making the fiber fit for weaving. Or making the thread for the sewing.

And you'd still need pants (tights or breeches) or a skirt, a bodice or vest, a jacket or cloak, stockings, and, if at all possible, but a rare luxury, shoes.

Back in the pre-industrial days, the making of thread, cloth, and clothing ate up all the time that a woman wasn't spending cooking and cleaning and raising the children. That's why single women were called "spinsters" — spinning thread was their primary job. "I somehow or somewhere got the idea," wrote Lucy Larcom in the 18th century, "when I was a small child, that the chief end of woman was to make clothing for mankind." Ellen Rollins: "The moaning of the big [spinning] wheel was the saddest sound of my childhood. It was like a low wail from out of the lengthened monotony of the spinner's life." (Jack Larkin, The Reshaping of Everyday Life, p. 26)

Anyway, with clothing that expensive and hard to make, every item was something you wore until it literally disintegrated. Even in 1800, a farm woman would be lucky to own three dresses: one for best and the other two for daily living. Heck, my mother, in 1930, went to college with that exact number of dresses to her name. This is why old clothing is rare: even the wealthy passed their old clothes on to the next generation or the poorer classes. The poor wore theirs until it could be worn no more, and then it was cut down for their children, and then used for rags of all kinds, and then, finally, sold to the rag and bone man who would transport it off to be made into (among other things) paper.



And speaking of paper, that was another thing that had to be invented for our society to exist: cheap paper. Good rag paper (made literally with expensive cloth rag) was always pricey, just not as pricey as parchment which was goat, sheep, or calf skin. (This is why medieval manuscripts were so few and why they were often kept chained up for fear of theft. It took at least a whole herd of animals to make the Book of Kells, for example. On the other hand, well-kept parchment can last thousands of years.) In fact, paper remained expensive long after clothing got cheaper, because it took a long time to figure out how to make paper out of nothing but wood pulp, without all that expensive rag content. It wasn't until the production of wood pulp paper was perfected in the mid-1800's that books (schoolbooks, fiction, non-fiction), magazines, and newspapers became available to the general public. Including pulp fiction — the first was Argosy Magazine in 1896 — a genre that was named for the cheapest of cheap fiber paper that it was published on. And without that pulp paper, where would our entire genre be? ■

Note: Check out the subsequent article "Is Time Money or is Money Time?" [hn.my/timemoney] which the author re-explained the idea of hours/time = money, with a couple of updates.

Eve Fisher is a retired history professor who still writes history articles, mystery stories, and the occasional rant. She lives in small town South Dakota with her husband and 5,000 books.

Reprinted with permission of the original author. First appeared in *hn.my/3500* (sleuthsayers.org)

Day in the Life of a Google Manager

By MATT WELSH

Not long after joining Google back in 2010, I wrote this cheeky piece [hn.my/googler] contrasting my daily schedule at Google with my previous career as an academic. Looking back on that, it's remarkable how much my schedule has changed in four years, in no small part because I'm now managing a team and as a result end up doing a lot less coding than I used to.

So, now seems like a good time to update that post. It will also help to shed some light on the differences between a pure "individual contributor" role and the more management-focused role that I have now.

By way of context: My role at Google is what we call a "tech lead manager" (or TLM), which means I'm responsible both for the technical leadership of my team as well as the people-management side of things. Our team has various projects, the largest and most important of which is the Chrome data compression proxy service. We're generally interested in making Chrome work better on mobile devices, especially for users in slow, expensive networks in emerging markets.

The best part of my job is how varied it is. Every day is different, and I usually have a lot of balls in the air. The below is meant to represent a "typical" day, although take that with a grain of salt given the substantial inter-day variation:

- **6:45am** Wake up. Get the kids up, get them ready, and make them breakfast. Shower.
- **8:30am** Jump on my bike and ride to work (which takes about 10 minutes), grab breakfast and head to my desk.
- **8:45am** Check half a dozen dashboards showing various metrics for how our services are doing: traffic is up, latency and compression are stable, datacenters are happily churning along.
- **9:00am** Catch up on email. This is a continuous struggle and a constant drain on my attention, but lately I've been using Inbox which has helped me to stay afloat. Barely.
- **9:30am** Work on a slide deck describing a new feature we're developing for Chrome, incorporating comments from one of the PMs. The plan is to share the deck with some other PM and Engineering leads to get buy-in and then start building the feature later this quarter.
- **10:00am** Chat with one of my teammates about a bug report we're chasing down, which gets me thinking about a possible root cause. Spend the next half hour running queries against our logs to confirm my suspicions. Update the bug report with the findings.
- **10:30am** I somehow find my morning has not been fully booked with meetings, so I have a luxurious hour to do some coding. Try to make some headway on rewriting one of our MapReduce pipelines in Go, with the goal of making it easier to maintain as well as adding some new features. It's close to getting done, but by the time my hour is up, one of the tests is still failing, so I will spend the rest of the day quietly fuming over it.

- **11:30am** Meet with one of my colleagues in Mountain View by video hangout about a new project we are starting up. I am super excited to get this project going.
- **12:00pm** Swing by the cafe to grab lunch. I am terrible about eating lunch at my desk while reading sites like Hacker News — some habits die hard. Despite this, I still do not have the faintest clue how Bitcoin works.
- **12:30pm** Quick sync with a team by VC to plan out the agenda for an internal summit we're organizing.
- **1:00pm** Hiring committee meeting. We review packets for candidates that have completed their interview loops and try to decide whether they should get a job offer. This is sometimes easy, but often very difficult and contentious, especially with candidates who have mixed results on the interview loop (which is almost everyone). I leave the meeting bewildered how I ever got a job here.
- **2:00pm** Weekly team meeting. This usually takes the form of one or more people presenting to the rest of the team something they have been working on with the goal of getting feedback or just sharing results. At other times we also use the meeting to set our quarterly goals and track how we're doing. Or, we skip it.
- **3:00pm** One-on-one meeting with a couple of my direct reports. I use these meetings to check in on how each member of the team is doing, make sure I understand their latest status, discuss any technical issues with their work, and also talk about things like career development, setting priorities, and performance reviews.
- **4:00pm** Three days a week I leave work early to get in an hour-long bike ride. I usually find that I'm pretty fried by 4pm anyway, and this is a great way to get out and enjoy the beautiful views in Seattle while working up a sweat.
- **5:00pm** Get home, shower, cook dinner for my family, do some kind of weird coloring or electronics project with my five-year-old. This is my favorite time of day.
- **7:00pm** Get the kids ready for bed and read lots of stories.
- **8:00pm** Freedom! I usually spend some time in the evenings catching up on email (especially after having skipped out of work early), but try to avoid doing "real work" at home. Afterwards, depending on my mood, might watch an episode of Top Chef with my wife or read for a while (I am currently working on Murakami's 1Q84).

Compared to my earlier days at Google, I clearly have a lot more meetings now, but I'm also involved in many more projects. Most of the interesting technical work is done by the engineers on my team, and I envy them; they get to go deep and do some really cool stuff. At the same time I enjoy having my fingers in lots of pies and being able to coordinate across multiple active projects, and chart out new ones. So it's a tradeoff.

Despite the increased responsibilities, my work-life balance still feels much better than when I was an academic. Apart from time-shifting some of my email handling to the evening (in order to get the bike rides in), I almost never deal with work-related things after hours or on the weekends. I have a lot more time to spend with my family and generally manage to avoid having work creep into family time. The exception is when I'm on pager duty, which is another story entirely: getting woken up at 3 am to deal with a crash bug in our service is always, um, exciting. ■

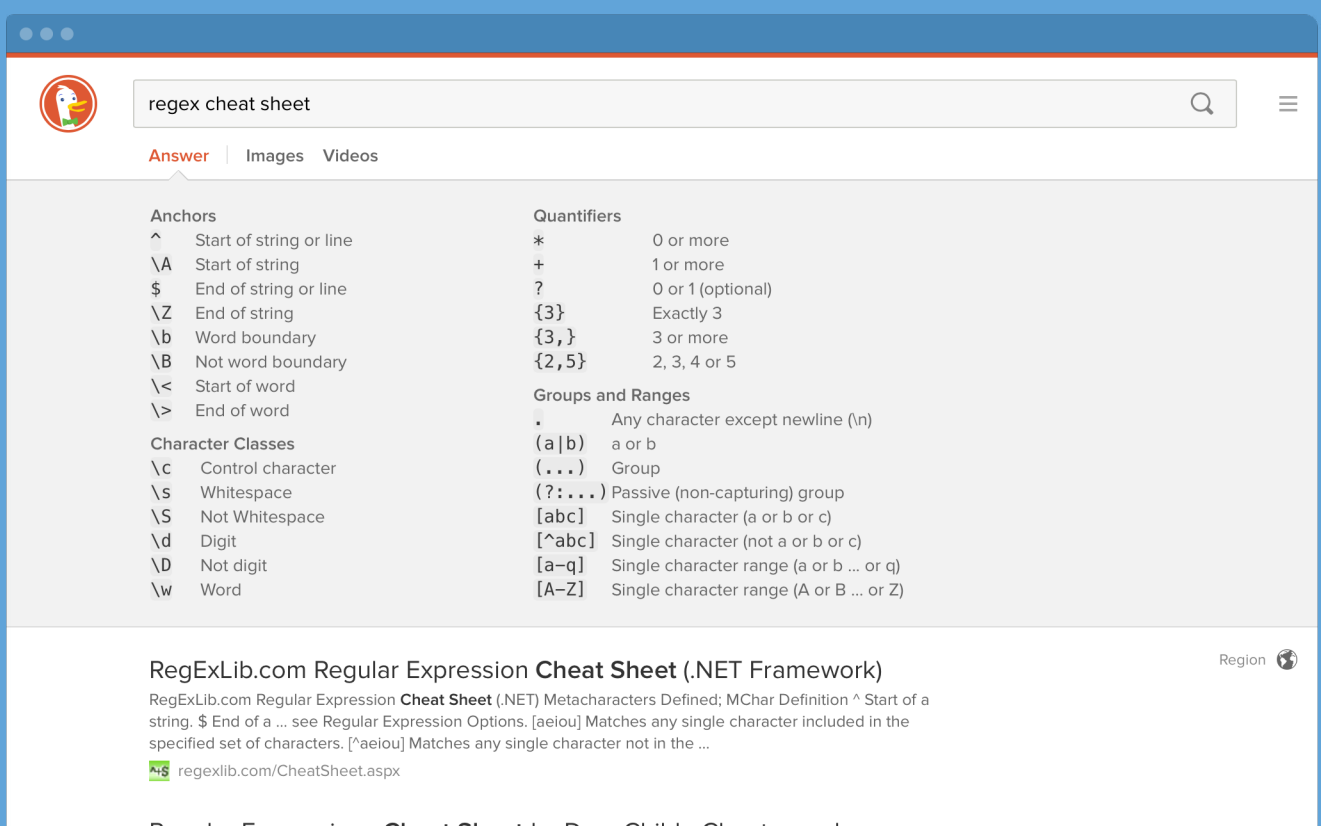
Matt Welsh is a software engineer at Google, where he works on mobile web performance. He was previously a professor of Computer Science at Harvard University. His research interests include distributed systems and networks.

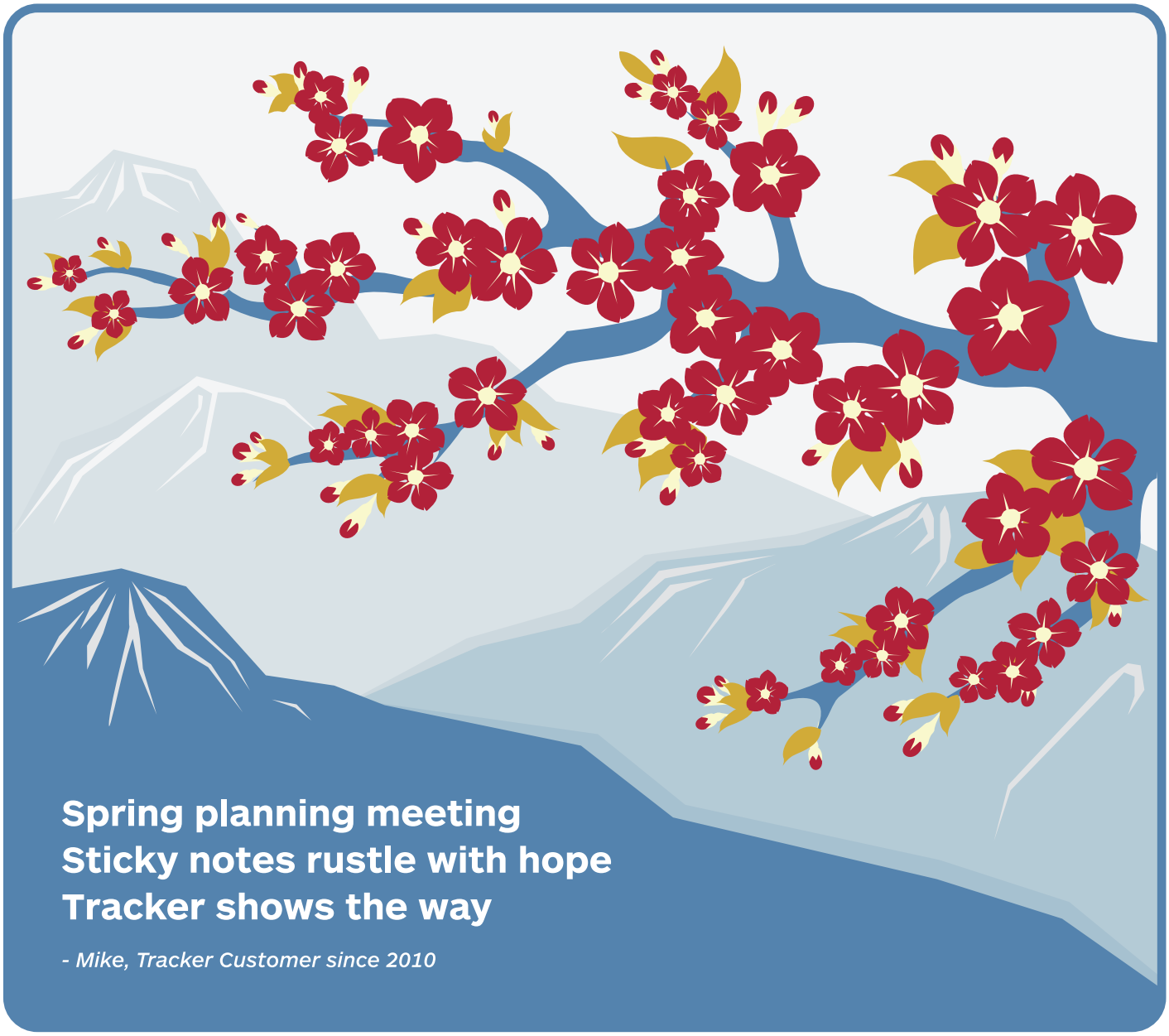
Reprinted with permission of the original author.
First appeared in *hn.my/gman* (matt-welsh.blogspot.com)

Join the DuckDuckGo Open Source Community.

Create Instant Answers or share ideas and help change the future of search.

Featured IA: Regex Contributor: mintsoft
Get started at duckduckhack.com





Discover the newly redesigned Pivotal Tracker

As our customers know too well, building software is challenging. That's why we created Pivotal Tracker, a pleasure-to-use project management tool, designed to facilitate constructive communication, keep teams focused, and reflect the true status of all your software projects.

With a new UI, cross-project funcionality, in-app notifications and more, staying zen in the face of looming business deadlines just got a little easier.

Sign up for a free trial, no credit card required, at pivotaltracker.com.

PivotalTracker

Build better software, faster.