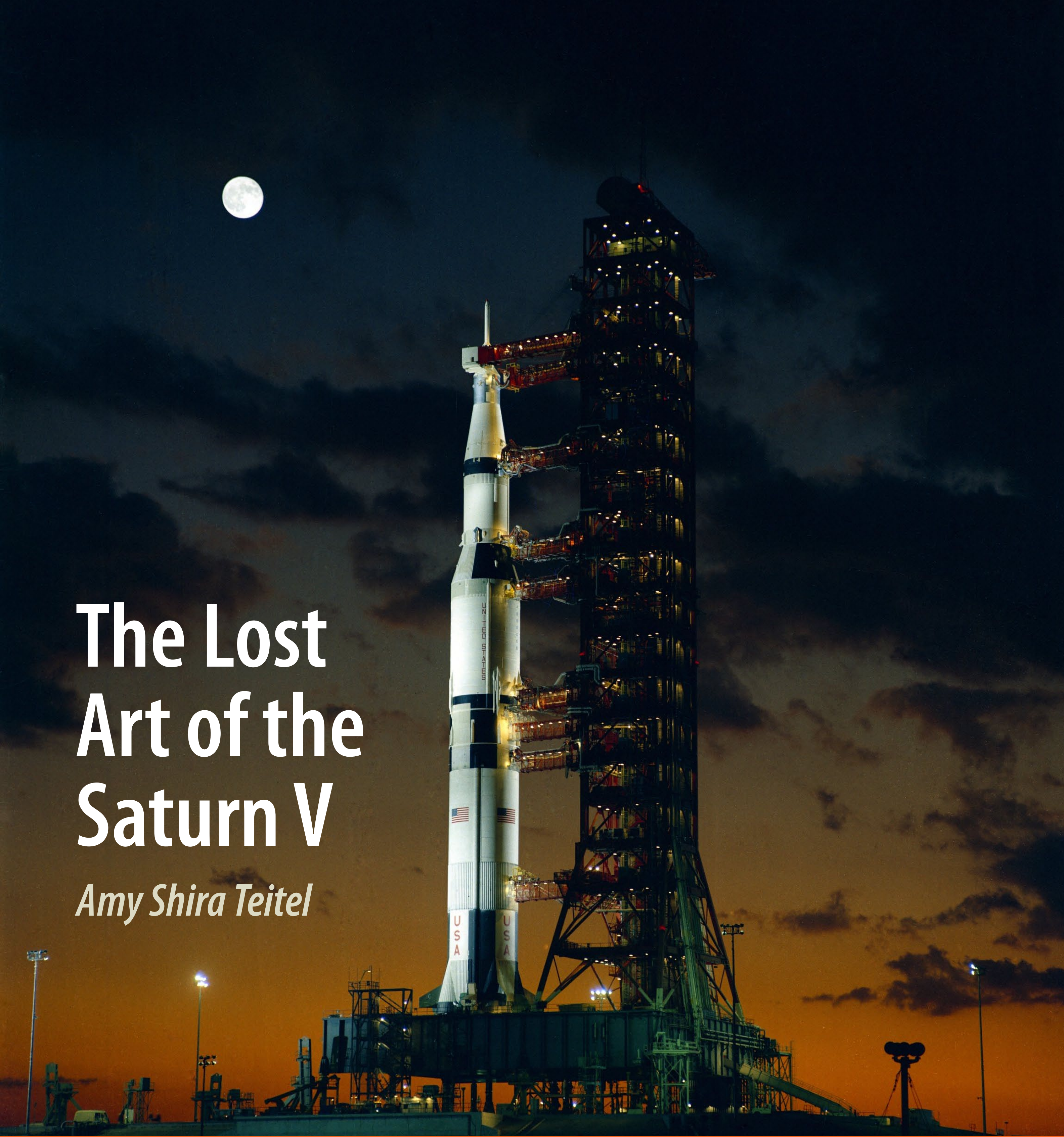


The Lost Art of the Saturn V

Amy Shira Teitel

HACKERMONTHLY

Issue 49 June 2014

HACK ON YOUR SEARCH ENGINE

and help change the future of search

duckduckhack.com

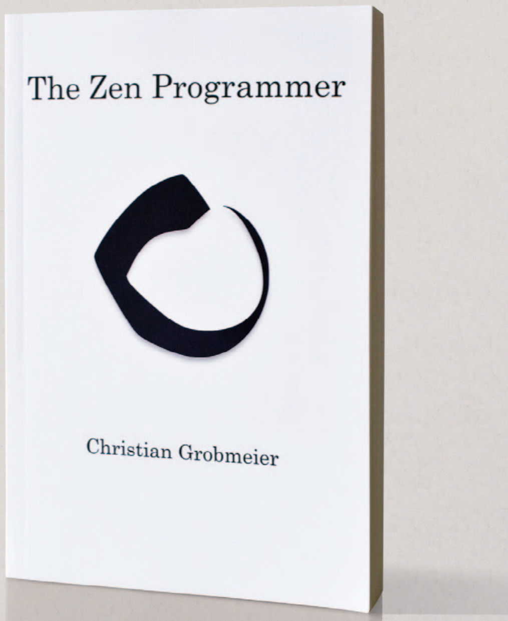
Declutter your mind ...

As featured in Hacker Monthly Issue 22

THE **ZEN** PROGRAMMER

There are more than just emails, phone calls and pressing issues – hackers often forget that. This book was written by a programmer who remembered a time in his life that didn't include a project manager dictating into his left ear on every little thing that needed to be completed. Just to calm him down, he started to meditate. After studying Zen for years, he wrote this book to help others escape from the hamster wheel.

Practice Zen at any time and anywhere. Read what Zen teachers knew about the modern day programmer, even hundreds of years ago!



Available in paperback and PDF.

For more information and samples, visit: www.zenprogrammer.org

Design: ahoj68.de

Curator
Lim Cheng Soon

Contributors
Amy Shira Teitel
Matt Jaynes
David Cramer
Peter Nixey
Gabriel Gonzalez
Amit Patel
James Hague
Gabriele Cirulli

Proofreaders
Emily Griffin
Sigmarie Soto

Ebook Conversion
Ashish Kumar Jha

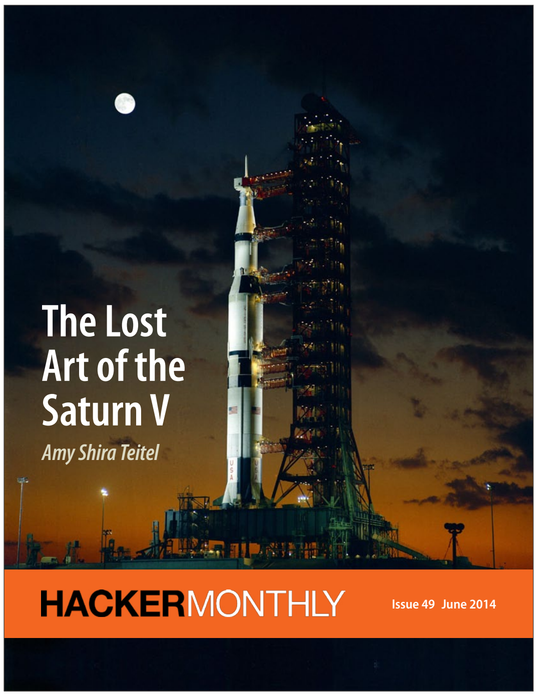
Printer
MagCloud

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

Advertising
ads@hackermonthly.com

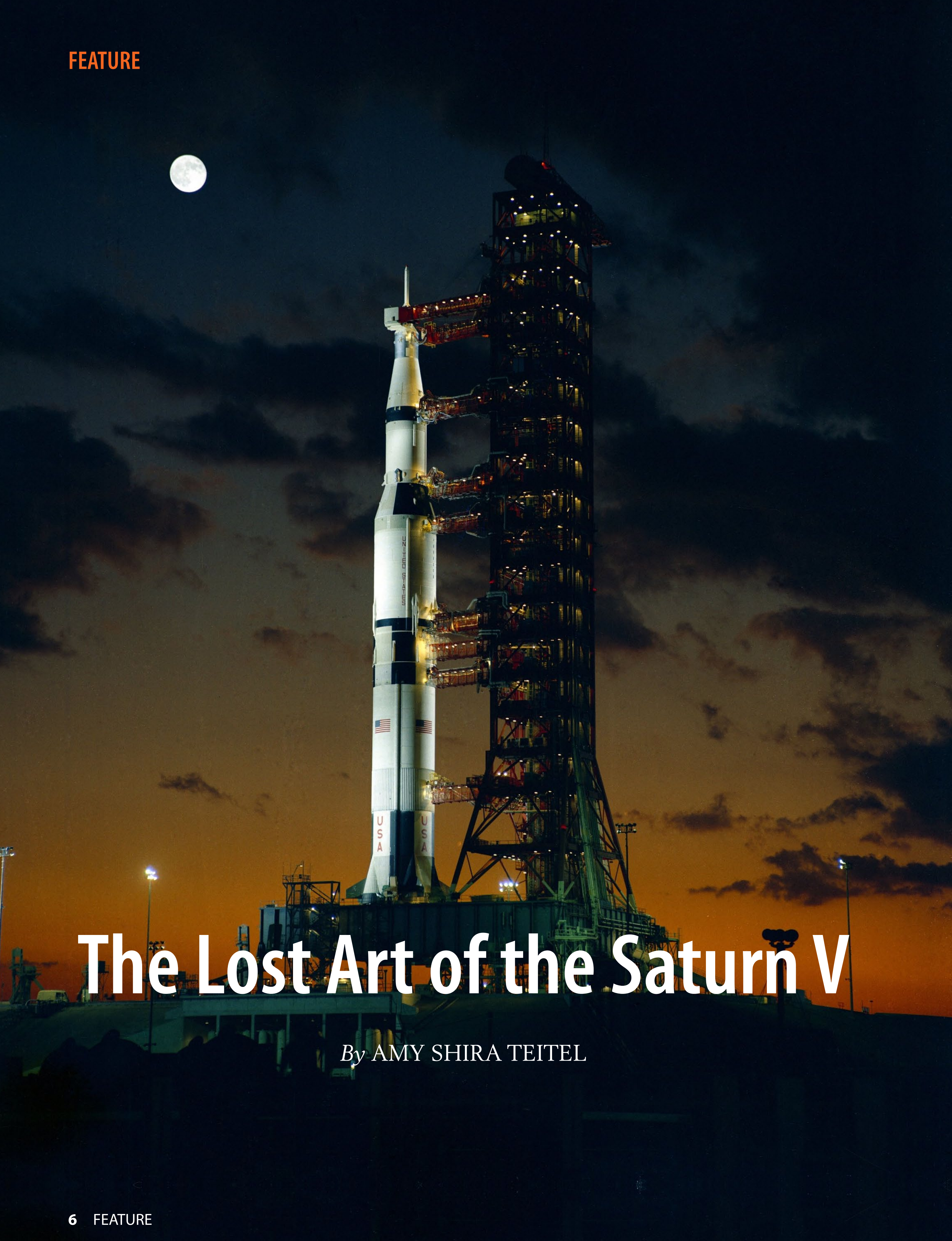
Contact
contact@hackermonthly.com

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



Contents

For links to Hacker News dicussions, visit *hackermonthly.com/issue-49*



The Lost Art of the Saturn V

By AMY SHIRA TEITEL

I'VE PREVIOUSLY MENTIONED that once the Shuttle program ends this year, there will be no way for NASA to launch manned missions. It simply doesn't have the necessary rockets to launch such a heavy payload into orbit, let alone a rocket capable of launching a heavy payload to another planet. A good example is the case of Mars. The Delta II hit its payload limit with the Mars Exploration Rovers Spirit and Opportunity, and that's with each rover launched separately. The upcoming Mars Science Laboratory rover Curiosity is significantly larger and will use an Atlas family launch vehicle. For NASA's Martian exploration plan to progress, as well as for the continuation of manned spaceflight, the organization needs a heavy lifting vehicle.

But NASA doesn't necessarily need a new launch vehicle. The organization had the means to launch a manned mission to Mars in the 1960s using only technology of the day. The whole mission, however, depended on the titanic Saturn V rocket, a technology that is lost to the current generation.

The Saturn V was the brainchild of Wernher von Braun, the man behind the Nazi V-2 missile that rained down on London in the final days of the Second World War. In 1945, with the Germans defeated and the Allies closing in to collect the brightest Nazi scientists as a form of intellectual reparations, von Braun and his team of rocketeers surrendered themselves to the Americans. They hoped their expertise in rocketry would be their ticket to continued work. It was; von Braun hand-selected 110 men to move to White Sands, New Mexico to join the Army Ballistic Missile Association (ABMA).

Wernher von Braun

The Jupiter C rocket launches Explorer 1. 1958.

The German rocketeers worked on developing improved missiles to launch the lightweight American warheads. But the Soviets soon proved the might of their rockets. The powerful R-7 launched the 182-pound Sputnik satellite, followed a month later by the 1,120-pound Sputnik II. The US was well behind in brute force lifting vehicles; the first successful US satellite was the 30-pound Explorer 1. The launch vehicle was the von Braun-designed, 69.5 foot high Jupiter C.

As the space race quickly picked up steam in the late 1950s, von Braun and his team in New Mexico found themselves with a new project: building a more powerful launch vehicle than anything the US currently had.

To this end, the rocketeers set to work in 1960 developing a new family of missiles named Saturn, the new rocket built on the successful Jupiter family of missiles and given the name of the next furthest planet from Earth. Their headquarters also received a new name. The ABMA became NASA's Marshall Spaceflight Centre (MSC) with von Braun as its first director.

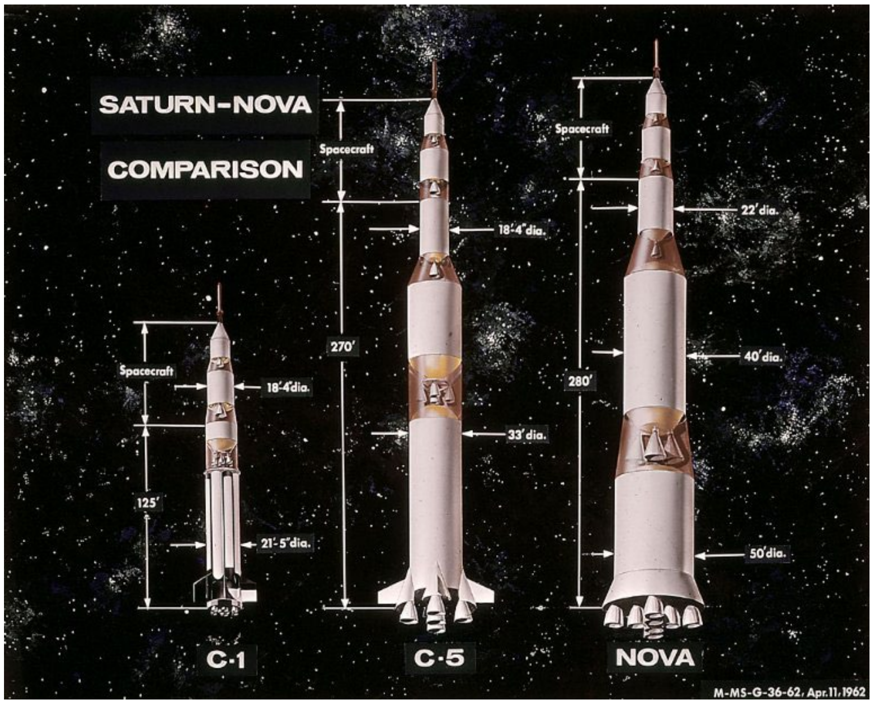
The new Saturn family of rockets was tied to NASA's long-term goals; though unofficial in 1960, the moon was an objective. But to build a rocket capable of sending men to the Moon, MSC engineers had to know how NASA intended to get there. There is more than one way to go to the Moon, and each decision requires different capabilities of its launch vehicle. In preparing Apollo, NASA considered three options called mission modes.

The first mode is the brute force method of direct ascent. A mammoth rocket is required to send a spacecraft on a straight path from the Earth to the Moon. The spacecraft would also have to land and relaunch from the Moon, making it heavy. Von Braun calculated that such a rocket would require around 12 million pounds of thrust at lift-off provided by eight engines. This method would require development of Nova, a missile of unparalleled power.

The second mode of Earth Orbit Rendezvous (EOR) uses two smaller rockets to assemble the same spacecraft in orbit around the Earth.

The third mode of Lunar Orbit Rendezvous (LOR) is the recognizable method that NASA used for the Apollo program. Two spacecraft would be launched on one powerful rocket, the lighter of which would land on the Moon while the heavier stayed in lunar orbit. This significantly lightened the payload and simplified the launch vehicle.

This method also brought an added safety measure to the lunar mission; it provided the astronauts with a stopping point in Earth orbit as well as lunar orbit. With more places to pause during a mission, there was more leeway to catch up on late maneuvers as well as a safe place to double check the mission profile. If any problems were detected, the crew could be brought home from Earth or lunar orbit much more easily than they could be from a lunar transit.



A comparison of Saturn and Nova launch vehicles. Research into Nova development was cancelled in 1962 when the LOR mission mode was selected. This artist's conception is from 1962.

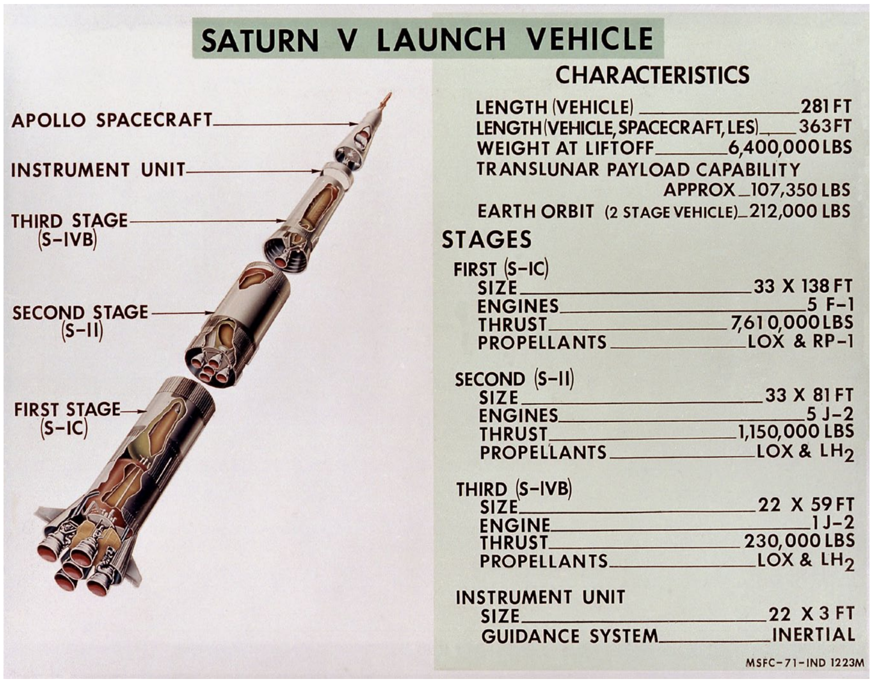
Within the developing Saturn family, only the Saturn V (so called as it was the fifth in the family) could launch the lunar spacecraft into Earth orbit then onto the Moon under its own power. At 364 feet tall, the three-stage rocket was the most powerful ever built.

The Saturn V's stages are the key to its power. The stages are stacked: the first stage on the bottom, the second stage on top of the first, and the third stage on top of the second. Above the third stage is the spacecraft. The stages burn and are discarded in sequence; as spent pieces of the rocket fall away, the payload headed towards the Moon became increasingly lighter and easier to lift.

The first stage (called the S-IC) provided the raw power. Two huge tanks, one containing 800,000 liters of refined kerosene the other 1.3 million pounds of other liquid oxygen (LOX), fuel five powerful engines. These engines produce 7.5 million pounds of thrust for about two and a half minutes, bringing the spacecraft to an altitude of about 38 miles. Once exhausted, the first stage falls away and the second stage takes over.

The second stage (called the S-II) burns for about six minutes, producing 1 million pounds of thrust from its five liquid hydrogen and LOX fuelled engines. The second stage shoots the spacecraft to an altitude of about 114 miles before it falls away exhausted.

The third stage (S-IVB) fires last and is responsible for propelling only the spacecraft. Its liquid hydrogen and LOX-fuelled engine fires twice; once for 2.75 minutes to bring the spacecraft to an altitude of 115 miles, and again for 5.2 minutes to initiate the lunar transit. With the final firing of the S-IVB, the Apollo crew is on their way to the Moon.



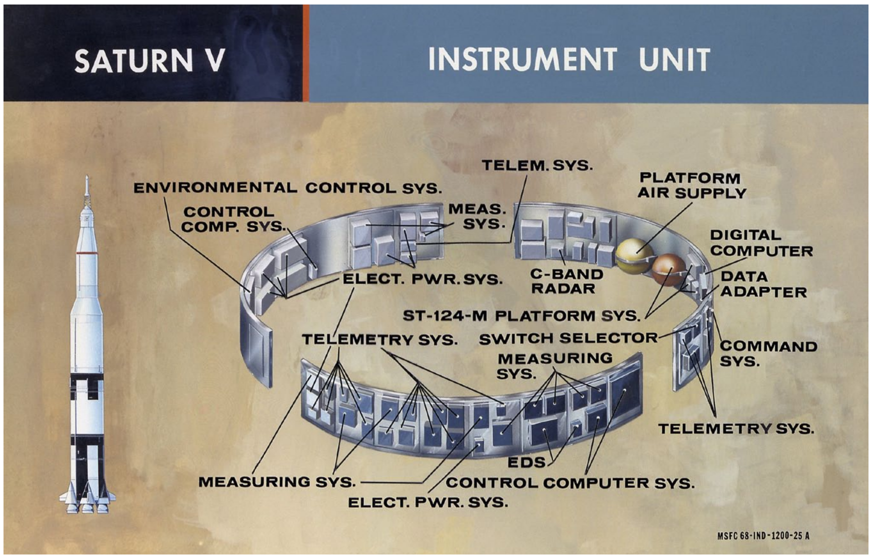
CHARACTERISTICS

LENGTH (VEHICLE)	281 FT
LENGTH (VEHICLE, SPACECRAFT, LES)	363 FT
WEIGHT AT LIFTOFF	6,400,000 LBS
TRANSLUNAR PAYLOAD CAPABILITY APPROX	107,350 LBS
EARTH ORBIT (2 STAGE VEHICLE)	212,000 LBS

STAGES

FIRST (S-IC)	
SIZE	33 X 138 FT
ENGINES	5 F-1
THRUST	7,610,000 LBS
PROPELLANTS	LOX & RP-1
SECOND (S-II)	
SIZE	33 X 81 FT
ENGINES	5 J-2
THRUST	1,150,000 LBS
PROPELLANTS	LOX & LH_2
THIRD (S-IVB)	
SIZE	22 X 59 FT
ENGINE	1 J-2
THRUST	230,000 LBS
PROPELLANTS	LOX & LH_2
INSTRUMENT UNIT	
SIZE	22 X 3 FT
GUIDANCE SYSTEM	INERTIAL

A cutaway of the Saturn V's stages.



A cutaway of the Saturn V's instrument unit.

While its three stages were responsible for the Saturn V's spectacular power, it wasn't the only factor that made it such a sophisticated launch vehicle. It also had a certain degree of autonomy. The brain of the Saturn V was its instrument unit, a ring of computerized components situated above the third stage. This included a digital computer, a stabilized guidance platform, and sequencers.

The rocket was able to guide itself into orbit and readjust its trajectory to achieve the orbital insertion point specified by the mission profile. Directional control was achieved through the first stages' engine configuration. The central engine was fixed, but the outer four were on gimbals and could swivel to direct the rocket's thrust in the desired direction.

This level of control was due to the rocket's inertial guidance system. Like the Apollo spacecraft, the Saturn V was aligned to the stars rather than any point on Earth. It used "fixed" stars to orient itself. The Saturn V's own guidance system wasn't only responsible for a successful orbital insertion; this was the guidance that shot the Apollo crew towards the Moon with the translunar injection or TLI burn.

The idea was that separating the rocket's computer and guidance systems from that of the spacecraft would provide an added redundancy. Apollo's onboard computer could control and steer the Saturn V. The Command Module (CM) was also aligned to the "fixed" stars for guidance with its own inertial guidance platform. But in the event Apollo's computer failed, NASA would have a potentially rogue Saturn V on its hands. With separate guidance systems, the crew was almost guaranteed a safe arrival into orbit at which point any problems could be addressed.

This proved to be a fortunate decision. When lightning struck Apollo 12 soon after launch (pictured), the CM's guidance system and computers were knocked offline. The Saturn V's systems, however, were unscathed. The crew and mission control were able to correct the problem in the spacecraft knowing they were still safely on course for orbit where an emergency abort and splashdown was simpler and safer.

The Saturn V's sophistication also makes it a complicated piece of technology. There are a lot of parts that have to function independently while simultaneously working together as a cohesive unit. And so von Braun, as the rocket's designer and director of the MSC, had to answer the same question that faced every new aspect of the space program: who would build it?

In the case of the Saturn V, the question was not only which subcontractor would build it, but how many. Should one contractor build the whole thing or should each stage be built by a different contractor? What about the instrumentation unit, the onboard computer, as well as the telemetry and radio systems? If each piece was made by a different contractor, who would oversee the final assembly and testing of the completed launch vehicle?

Von Braun made the decision to give each piece of the rocket to a different contractor, a decision that yielded mutual gain. From the contractors' standpoint, multiple companies were able to benefit financially as well as partake in the challenge of building the Saturn V. From von Braun's perspective, it enabled him to pull together the best in the industry; the top men from each company worked towards building his launch vehicle.

Three main companies were awarded Saturn V contracts. Boeing built the first stage, North American Aviation (who built the X-15 and the Apollo CM) built the second stage, and Douglas Aircraft built the third stage. The inertial guidance system and instrumentation was built in-house by the Marshall Spacecraft Centre — it made sense to keep the brains of the rocket close to the men who would control it during a launch.

To simplify the oversight of proceedings around the Saturn V's construction, von Braun created two groups within the MSC. The Research and Development Operations team became the architects overseeing the rocket's integrity and structure, and the Industrial Operations team funded and oversaw the subcontractors.

The Saturn V was completed at an impressive speed. Construction began in 1960. Each element was tested individually before the first launch of a complete Saturn V in 1967, which launched an unmanned Apollo CSM as payload. There was no time to waste a launch using a dummy spacecraft or a water tank as ballast. Everything had to advance the goal of the lunar landing.

After only two unmanned launches, the third Saturn V took Apollo 8 to the Moon.

The Saturn V fell out of favor with NASA in the mid-1970s; Apollo was no longer a viable program and NASA had begun to favor the reusable low Earth orbital space shuttle. There were no immediate plans to return to the Moon or any foreseeable need for such a powerful launch vehicle. In the intervening nearly 40 years, the technology behind the Saturn V has been all but lost.

The launch of Apollo 8. 1968.

The division of labor on the Saturn V's construction proved, in retrospect, to be a double-edged sword. On the one hand, it allowed the rocket to be completed at an incredible rate, certainly responsible for the success of the Apollo program.

But on the other hand, building the rocket at such a rate and with so many subcontractors means the people who oversaw and understood the actual assembly and overall working of the Saturn V were few. Each contractor recorded the workings of their stage and records survive about the engines used, but only a handful of engineers from the MSC knew how the Saturn V puzzle fit together.

Composite image of every Saturn V launch. Apollo 5 and Apollo 7 were launched on the slightly smaller Saturn IVB launch vehicle.

It is possible to work backwards to recreate individual aspects of the technology, but the men who knew how the whole vehicle worked are gone. No one alive today is able to recreate the Saturn V as it was.

Worse is the lack of records. Without a planned used for the Saturn V after Apollo, most of the comprehensive records of the rocket's inner workings stayed with the engineers. Any plans or documents explaining the inner workings of the completed rocket that remain are possibly living in someone's basement, unknown and lost in a pile of a relative's old work papers.

Two Saturn Vs remain today as museum pieces, but it is likely that the rocket will never see a rebirth and reuse in manned spaceflight.

Yes, NASA put men on the Moon with 1960s technology, but that technology doesn't exist anymore. By default, neither does the possibility of a manned lunar or Martian mission for that matter without a new launch vehicle. A new heavy lifting vehicle will eventually come about — it will have to for NASA to pursue its longer-term goals. Until then, NASA is bound to low Earth orbit and minimal interplanetary unmanned spacecraft. ■

Amy Shira Teitel is a spaceflight historian and freelance writer fascinated with the early history of the space age, particularly the unflown technologies, unrealized programs, and all the details that never make it into the popular histories. She writes, among other sites, for Motherboard, DVICE, Discovery News, and her blog, Vintage Space, is hosted at Popular Science.

Boring Systems Build Badass Businesses

By MATT JAYNES

LET ME TELL you a story about systems do's and don'ts and how they relate to business success. Of all the mistakes I see businesses make, this is one of the most common. It's a critical failing that cripples or kills many businesses that could have otherwise been successful.

Background

Alice and Zola were rivals who both had dreams of building their own restaurant empires.

They each applied for and won $1 million grants to open their restaurants — yay!

Alice's Build

Alice spent $500K to build a large restaurant and hired a handyman named Albert to lead the effort.

Albert was one of the most creative and smartest handymen in the world. Alice quizzed him directly from the manuals of all the top plumbing and electrician books and he passed with flying colors!

So, when designing the plumbing and electrical systems for the restaurant, Albert chose all the most exciting and cutting edge technologies!

He put a different brand of plumbing system in each section of the restaurant because each area had slightly different needs. One system went in the bathrooms, the second went in the kitchen, the third went in the lobby, and the fourth went outside.

He was even more innovative with the electrical systems — and put in a total of 10 different systems throughout the restaurant.

They were now 6 months behind schedule, but Alice now had the restaurant with the most innovative plumbing and electrical systems in the whole country. So, naturally Alice and Albert busted open the champagne to celebrate!

Zola's Build

Zola's path to launch was a bit different.

She also spent $500K to build a large restaurant but hired a handyman named Zip to lead the effort.

Zip had a reputation for building simple systems that required little maintenance and just worked. Zola hired him based on his track record and they got to building.

When choosing the electrical and plumbing systems, Zip just chose the industry standard systems that had been around for years. These systems had great manuals and great companies backing them with plentiful support and spare parts.

Since they chose simple standard systems, they got done 2 months early and even had money left over to create a gorgeous atrium they knew the customers would love.

Alice's Run

When Alice's restaurant finally opened everything went great! Well, until the lunch rush. Then the power went out.

Albert spent the next 2 days without sleep trying to track down the problem. It turned out that the fancy electric toilets were used too frequently during the rush and burned out the relays in 4 of the 10 electrical systems.

Over the next 3 months the restaurant would open for a few days, then close to deal with some technical problem. Albert would heroically work nights and weekends to solve the problem so that the restaurant could stay open at least some of the time.

Alice was sooo grateful she had hired Albert since he was super smart and could always eventually figure out and fix even the toughest problems with the systems. In Alice's eyes, Albert was a real hero to work overtime to fix the problems.

However, Albert eventually got burned out and bored with Alice's restaurant, so he left. He figured all the problems were just bad luck — maybe next time he'd be luckier!

Alice now had to try to hire a replacement for Albert. The reputation of her restaurant was very poor now, so it was difficult to find applicants. Finally she found someone willing to take the job. Unfortunately he couldn't figure out the complex interactions between the systems since Albert hadn't left any notes. Alice hired more and more technicians to try and figure out the systems. Eventually after hiring 10 full-time technicians, they were able to figure out the systems and get them working again after a few months.

During that time, they discovered that 2 of the electrical systems and 1 of the plumbing systems had been abandoned by their creators and there was no longer support or parts for those systems. So, Alice had to hire 2 more technicians to support these now defunct systems.

All these technicians ate up the remaining money she had and made it impossible for her to ever get cash-flow positive.

The restaurant went bust and Alice decided to apply to grad school.

Zola's Run

Since the plumbing and electrical systems just worked, Zola was able to put all her focus into hiring great chefs, great entertainers, and great serving staff. She was able to innovate and come up with new exciting events and dishes for her dining guests.

Zip was rarely needed. He once fixed a cracked pipe, but it only took him 5 minutes. After a couple months he got another job and moved out of state.

Zola quickly found Zed as a replacement. He was eager to work there because of Zola's reputation and because he was very familiar with the standard systems they used.

Her restaurant's reputation grew every day and so did the demand to eat there. Soon there was nearly always an hour's wait to get in.

She still had $400K left from the grant and had earned another $1.2 million over the last year. With all that cash, she was able to start her true restaurant empire by opening another 2 restaurants.

Extreme?

This may seem like an extreme story - but I've seen much more drastic outcomes in the tech space.

I had a front-row seat to watch a company spend $14+ million on a system that was so complex and buggy it was eventually abandoned as a complete loss. In contrast, I was there when a startup scaled to over 100 million users with just a couple good engineers with simple standard systems.

If you follow tech news, you'll have heard of even more extreme scenarios — where the losses or wins were in the billions.

Common Objections

This makes sense for the underlying systems, but what about development of the actual products?

Build the most minimal solution you possibly can. See if customers will like it, use it, and pay enough for it. Only then build it into a full solution. Simplicity, great test coverage, and great documentation will ensure what you build retains its value long-term. You'll save a ton of time and money going this route which you can then use to create even more profitable products for your customers. Always be asking "How can I do this faster, simpler, cheaper?"

But don't you want your developers to be engaged and working on interesting projects?

If your developers are desperate to play with novel technologies - just give them more time off work to play with their own projects. Google, 37Signals, and GitHub have all done this to great benefit. There are many ways to achieve developer happiness, but making your core business products a playground for developers seeking novelty is the path to hell.

But [some new unproven system] is really cool! Even [some big company] uses it!

Great! Then play with it to your heart's delight. However, do it on your own time. Don't jeopardize your business with it. Do you care more about playing with novel technologies than spending your energy and innovation on the products your customers actually care about? Remember, you're a business, not a college R&D lab.

But [some company] I know used a ton of crazy cool new tech and still got acquired for millions!

I've certainly seen this happen. However, often those companies are acquired for much less than they could have been and frequently dissolve once they've been bought. I worked for a startup that made these mistakes and lost ~$2 million due to it (buying $1 million of cool hardware they didn't need, hiring awesome data warehousing engineers when there were no data warehousing needs, etc). They still got acquired, but for probably 1/3 of what they could have been if they had spent that lost money on marketing and a better product. Within a year, the acquirer realized it had purchased a huge mess and dissolved the acquired company. Tens of millions of dollars down the drain.

Avoid the Pitfalls

In my contracting career, I've seen the inner workings of many different companies. Here are a couple rules to avoid the most common mistakes I see:

- Innovate on your core product, not on your plumbing (this rule is extremely tempting for developers to break — see next rule)
- Choose developers based on their track record and their commitment to ruthless simplicity and business growth

In the end, your business exists to create business value, not be a plumbing showcase. ■

Matt Jaynes is a systems engineer. He recently founded DevOps University to help developers learn how to build and manage their own systems.

Metrics and monitoring for people who know what they want

We know from experience that monitoring your servers and applications can be painful, so we built the sort of service that we would want to use. Simple to set up, responsive support from people who know what they're talking about, and reliably fast metric collection and dashboards.

Dashboards **StatsD** **Happiness**

Why Hosted Graphite?

- **Hosted metrics and StatsD:** Metric aggregation without the setup headaches
- **High-resolution data:** See everything like some glorious mantis shrimp / eagle hybrid*
- **Flexibile:** Lots of sample code, available on Heroku
- **Transparent pricing:** Pay for metrics, not data or servers
- **World-class support:** We want you to be happy!

Promo code: **HACKER**

Grab a free trial at http://www.hostedgraphite.com

*Hosted Graphite's mantis shrimp / eagle breeding program has been unsuccessful thus far

HOSTEDGRAPHITE

Scaling SQL with Redis

By DAVID CRAMER

I LOVE REDIS. IT'S one of those technologies that is so obvious it makes you wonder why it took so long for someone to build it. Predictable, performant, and adaptable, it's something I've come to use more and more over the last few years. It's also no secret that Sentry is run primarily on PostgreSQL (though it now also relies on a number of other technologies).

A little over a week ago I gave a keynote at Python Nordeste. At some point it was suggested I give a lightning talk. I decided I'd talk about some of the cool hacks we use to scale Sentry, specifically with Redis. This article is an expanded version of that five-minute talk.

Alleviating Row Contention

Something we adopted early on in Sentry development was what's become known as `sentry.buffers`. It's a simple system which allows us to implement very efficient buffered counters with a simple **Last Write Wins** strategy. It's important to note that we completely eliminate almost any form of durability with this (which is very acceptable for the way Sentry works).

The operations are fairly straightforward, and whenever an update comes in we do the following:

1. Create a hash key bound to the given entity
2. Increment "counter" using `HINCRBY`
3. `HSET` any various `LWW` data (such as "last time seen")
4. `ZADD` the hash key to a "pending" set using the current timestamp

Now every tick (in Sentry's case, this is 10 seconds) we're going to dump these buffers and fanout the writes. This looks like the following:

1. Get all keys using `ZRANGE`
2. Fire off a job into RabbitMQ for each pending key
3. `ZREM` the given keys

Now the RabbitMQ job will be able to fetch and clear the hash, and the "pending" update has already been popped off of the set. There are a few things to note here:

- We use a sorted set for the case where we would want to only pop off a set amount (e.g., we want to process the 100 oldest).
- If we end up with multiple queued jobs to process a key, one could get no-oped due to another already processing and removing the hash.
- The system scales consistently on many Redis nodes simply by putting a "pending" key on each node.

With this model we mostly guarantee that only a single row in SQL is being updated at once, which alleviates most locking contention that we'd see. This greatly benefits Sentry given that it might deal with a burst of data that all ends up grouping together into the same counter.

Rate Limiting

Due to the nature of Sentry we end up dealing with a constant denial-of-service attack. We've combatted this with a number of rate limiters, one of which is powered by Redis. This is definitely one of the more straightforward implementations and lives within `sentry.quotas`.

The logic is very straightforward, and looks something like this:

```
def incr_and_check_limit(user_id, limit):
    key = '{user_id}:{epoch}'.format(user_id,
int(time() / 60))

    pipe = redis.pipeline()
    pipe.incr(key)
    pipe.expire(key, 60)
    current_rate, _ = pipe.execute()

    return int(current_rate) > limit
```

The way we do rate limiting illustrates one of the most fundamental benefits of Redis over memcache: incr's on empty keys. To achieve the same behavior in memcache would likely end up with this kind of approach:

```
def incr_and_check_limit_memcache(user_id,
limit):
    key = '{user_id}:{epoch}'.format(user_id,
int(time() / 60))

    if cache.add(key, 0, 60):
        return False

    current_rate = cache.incr(key)

    return current_rate > limit
```

We actually end up employing this strategy on a few various things within Sentry to do short-term data tracking. In one such case we actually store the user's data in a sorted set so we can quickly find the most active users within a short time period.

Basic Locks

While Redis isn't highly available, our use case for locks makes it a good tool for the job. We don't use them in Sentry's core anymore, but an example use case was where we wanted to minimize concurrency and to simply no-op an operation if something appeared to be running already. This can be useful for cron-like tasks that may need to execute every so often, but don't have strong coordination.

In Redis, doing this is fairly simple using the `SETNX` operation:

```
from contextlib import contextmanager

r = Redis()

@contextmanager
def lock(key, nowait=True):
    while not r.setnx(key, '1'):
        if nowait:
            raise Locked('try again soon!')
        sleep(0.01)

    # limit lock time to 10 seconds
    r.expire(key, 10)

    # do something crazy
    yield

    # explicitly unlock
    r.delete(key)
```

While the `Lock()` within Sentry makes use of memcached, there's absolutely no reason we couldn't switch it over to Redis.

Time Series Data

Recently we wrote a new mechanism for storing time-series data in Sentry (contained in `sentry.tsdb`). This was heavily inspired by RRD models, specifically Graphite. We wanted a simple and fast way to store short-period (e.g. one month) time-series data that would allow us to handle very high throughput for writes, and allow us extremely low latency for computing short-term rates. While this is the first model where we actually want to persist data in Redis, it's yet another simple case of using counters.

Our current model stores an entire interval's series within a single hash map. For example, this means all counts for a given key-type and for a given 1-second live in the same hash key. It looks something like this:

```
{
    "<type enum>:<epoch>:<shard number>": {
        "<id>": <count>
    }
}
```

So in our case, let's say we're tracking the number of events. Our enum maps the Event type to "1". The resolution is 1s, so our epoch is simply the current time in seconds. The hash ends up looking something like this:

```
{
    "1:1399958363:0": {
        "1": 53,
        "2": 72,
    }
}
```

An alternative model might just use simple keys and just perform `incrs` on those buckets:

```
    "1:1399958363:0:1": 53
```

We chose the hash map model for two reasons:

- We can TTL the entire key at once (this also has negative side effects, but so far has been stable).
- The key gets greatly compressed, which is a fairly significant deal.

Additionally, the shard number key allows us to distribute a single bucket over a fixed number of virtual shards (we use 64, which map to 32 physical nodes).

Now querying the data is done using Nydus [hn.my/nydus] and its `map()` (which relies on a worker pool). The code is pretty hefty for this operation, but hopefully it's not too overwhelming:

```
def get_range(self, model, keys, start, end, rollup=None):
    """
    To get a range of data for group ID=[1, 2, 3]:

    Start and end are both inclusive.

    >>> now = timezone.now()
    >>> get_keys(tsdb.models.group, [1, 2, 3],
    >>>          start=now - timedelta(days=1),
    >>>          end=now)
    """
    normalize_to_epoch = self.normalize_to_epoch
    normalize_to_rollup = self.normalize_to_rollup
    make_key = self.make_key

    if rollup is None:
        rollup = self.get_optimal_rollup(start, end)

    results = []
    timestamp = end
    with self.conn.map() as conn:
        while timestamp >= start:
            real_epoch = normalize_to_epoch(timestamp, rollup)
            norm_epoch = normalize_to_rollup(timestamp, rollup)

            for key in keys:
                model_key = self.get_model_key(key)
                hash_key = make_key(model, norm_epoch, model_key)
                results.append((real_epoch, key, conn.hget(hash_key, model_key)))

            timestamp = timestamp - timedelta(seconds=rollup)

    results_by_key = defaultdict(dict)
    for epoch, key, count in results:
        results_by_key[key][epoch] = int(count or 0)

    for key, points in results_by_key.iteritems():
        results_by_key[key] = sorted(points.items())
    return dict(results_by_key)
```

It boils down to the following:

- Generate all of the required keys.
- Using the worker pool, fetch all of the results in the minimum number of network operations (Nydus takes care of this).
- Given the results, map them to a result set that represents the buckets based on the given intervals, and the given keys.

Simple Choices

I'm a huge fan of simple solutions to problems, and Redis definitely fits in that bucket. Its documentation [redis.io/commands] is amazing, and it's the lowest barrier of entry you're going to find outside of something like memcached. While it has its tradeoffs (primarily if you're using it with persistence), they're up front and fairly straightforward.

What can Redis solve for you? ■

David is the founder of Sentry and works in engineering at Dropbox. He's an active member of the open source community, passionate about scale, simplicity, and usability

How To Be A Great Software Developer

By PETER NIXEY

IF THERE'S ONE thing that software developers care about, it's becoming even better software developers. Where do you start though? Should you accumulate the bells and whistles: deepen your knowledge of Node and no-sequel? Should you rote-learn the answers to the profession's gateway questions and be able to produce bubble sort or link shortener algorithms on demand? Or are there perhaps more fundamental roots that you can put down?

I believe that your seniority and value as a programmer is measured not in what you know, it's measured in what you put out. The two are related but definitely not the same. Your value is in how you move your project forward and how you empower your team to do the same. In fifteen years of programming I've never had to implement a bubble sort or a link shortener. However I have had to spend thousands and thousands of hours writing and refactoring account management tools, editing suites, caching logic, mailing interfaces, test suites, deployment scripts, JavaScript layers, analytics architecture and documentation. These were the things that mattered, the completion of these were what moved us forward.

Those humble components are the bricks and mortar of projects and take hundreds or thousands of hours of hard work to assemble. And even though they combine to form complex systems, they themselves should not be complicated.

You should aim for simplicity, and over the years I have learned that simplicity is far more easily attained by time spent working and refactoring than hours of pure thought and "brilliance."

Simplicity and excellence are most reliably attained by starting with something, anything, that gets the job done and reworking back from that point. We know this is true of companies and the concept of the MVP is burned deep into our consciousness. So, too, with software. Start with something ugly but functional and then apply and reapply yourself to that ugly and misshapen solution and refactor it back into its simplest form. Simplicity comes far more reliably from work than from brilliance. It comes more predictably from code written, than from thought expended. It comes from effort.

Your value as a developer is measured not in the height of your peaks, but the area under your line.

It is all too easy for smart lazy people to flash spikes of brilliance and wow their contemporaries, but companies are not built on those people and product does not sit well on spikes. Companies are built on people and teams who day in, day out, commit good code that enables others do the same. Great product is built by work horses, not dressage horses.

Years after Joel coined the term "Rockstar Programmer," it lives on along with the misapprehension that companies need such geeky micro-celebrities in order to do anything. While those characters do exist there aren't many of them. When you do find them they're often erratically brilliant — astonishing at the things that interest them but hopeless at working consistently or smoothly with their team.

Not only is their output erratic but their superiority is aspirational and infectious. Their arrogance bleeds toxically into the rest of the team. It signals loud and clear that if you're smart enough you choose when you work and what you work on. You become a "Developer in Residence." And you not only soak up a salary but you distort the values of those who work around you.

So the reality is that in all likelihood you and your team will depend, should depend not on those who think they are "Rockstars" or "Ninjas" but on reliable people who work in reliable ways.

Great developers are not people who can produce bubble sorts or link shorteners on demand. They are the people who when you harness them up to a project, never stop moving and inspire everyone around them to do the same. Fuck Rockstars. Hire workhorses. Here are some ways to become one:

Name your functions and variables well (write Ronseal Code)

Such an incredibly simple place to start, and yet I think it is one of THE most important skills in programming. Function naming is the manifestation of problem definition which is frankly the hardest part of programming.

Names are the boundary conditions on your code. Names are what you should be solving for.

If you name correctly and then solve for the boundary conditions that that name creates, you will almost inevitably be left with highly functional code.

Consider the function:

```
def process_text string
  …
end
```

It tells someone almost nothing about what it's going to do or how it's been implemented in the code. However:

```
def safe_convert_to_html string
  ...
end
```

tells someone exactly what's going to happen. It's also a good indicator as to what's not going to happen. It tells you both what you can expect the method to do but also how far you can overload that method.

A developer might happily refactor "process_text" to not only convert text to HTML but to auto-embed videos. However that may be resolutely not what was expected in some of the places that function was used. Change it and you've created bugs. A good clear name is a commitment to not just what a function does but also what it won't do.

Function names create contracts between functions and the code that calls them. Good naming defines good architecture.

A good function promises what it will deliver and then delivers it. Good function and variable naming makes code more readable and tightens the thousands of contracts which crisscross your codebase. Sloppy naming means sloppy contracts, bugs, and even sloppier contracts built on top of them.

It's not just functions that you can leverage to describe your code. Your variable names should also be strong. Sometimes it can even be worth creating a variable simply in order to document the logic itself.

Consider the line:

```
if (u2.made < 10.minutes.ago)
   && !u2.tkn
   && (u2.made == u2.l_edit)
  ...
```

It's pretty hard to figure out what the hell is happening there, and even once you have done so, it's not 100% clear what the original author was trying to achieve with it. The variable names are horrible and tell you nothing.

The "and not" statement is always confusing to read (please never write "and not" clauses which end with a noun), and if your job was to refactor this code, you'd have to do some acrobatics to guess exactly what the original intent was.

However, if we change the variables names into something more meaningful then things immediately start to become clearer:

```
if (new_user.created_at <
10.minutes.ago)
  && !new_user.invitation_token
  && (new_user.created_at ==
new_user.updated_at)
```

We can go further still and forcibly document the intent of each part of the if statement by separating and naming the components:

```
user_is_recently_created =
user.created_at < 10.minutes.
ago
invitation_is_unsent = !user.
invitation_token
user_has_not_yet_edited_pro-
file = (user.created_at == user.
updated_at)

if user_is_recently_created
  && invitation_is_unsent
  && user_has_not_yet_edited_
profile
  ...
```

It takes some courage to write a line like "user_is_recently_created" because it's such fuzzy logic, but we all do it at times, and owning up to that helps inform the reader about the assumptions you've made.

Notice also how much stronger this approach is than using comments. If you change the logic there is immediate pressure on you to change the variable names. Not so with comments. I agree with DHH, comments are dangerous and tend to rot — much better to write self-documenting code.

The better code describes itself, the more likely someone will implement it the way it was intended and the better their code will be. Remember, there are only two hard problems in computer science: cache invalidation, naming, and off-by-one errors.

If you want to be a great developer, make sure you write Ronseal Code that does exactly what it says on the tin.

Go deep before you go wide: learn your chosen stack inside out

Very few programming problems are genuinely new. Very few companies are doing technical work that hasn't already been done by 50 teams before them. Very few problems attract Stack Overflow eyeballs that haven't already seen them somewhere else before.

For that exact reason, the majority of the things you are trying to do have already been solved by the very stack you're already using. I once refactored 60 lines of someone else's Rails code to one line using the delightfully simple and powerful methods that Rails ships with.

Most programmers waste huge amounts of time by lazily re-creating implementations of pre-existing functionality.

Not only do they waste time but they create verbosity and errors. Their code requires new documentation to describe it, new tests to monitor it, and it makes the page noisier and harder to read. Like any new code, it's also buggy. War-tested (and actually-tested) stack code is very seldom buggy.

If you are a Ruby developer, take time to learn Ruby, especially the incredible range of array methods. If you are a Node developer, take time to understand the architecture, the methods and the mindset of Node. If you are an Angular developer, go right up to the rock-face and understand the logic behind of the incredible architecture the core team is forging there right now. Ask before you invent. You are walking in the shadows of giants. Take time to find their tracks and then marvel at how beautifully they have been built. Because if you don't, you simply punt the problem downstream and someone will just have to figure out why the hell you chose the sub-standard path you did.

Learn to detect the smell of bad code

Something I've noticed in programmers who are good but who have plateaued is that they simply don't realize that their code could be better. That is one of the worst things that can happen to your personal development. You need to know what has to improve before you can figure out how to improve it. Learn both what good code looks like and what bad code looks like. It is said that grand chessmasters spend proportionally much more time studying previous other good chess player's games than the average players. I'm quite certain that the same is true for top developers.

An important part of your improvement arsenal is your ability to detect bad code — even when it's only slightly bad or perhaps "a bit smelly." Smelly code is code which, while you can't quite articulate why, just doesn't feel right.

It may be that you've used 60 lines of code for something which feels fundamentally simpler; it might be something which feels like it should be handled by the language but has been manually implemented instead; it might just be code that is complicated as hell to read. These are your code smells.

It's not an easy thing to do, but over the years you should learn what bad code smells like and also what beautiful code looks like. You should develop an aesthetic appreciation for code. Physicists and mathematicians understand this. They feel very uneasy about an ugly theory based on its ugliness. Simplicity is beautiful and simplicity is what we want.

The truth is that the truth is sometimes ugly, but you should always strive for beauty. And when ugly is the only way, know how to present it well. If you can't create beautiful code, at least create Shrek code, but before you do either you need to develop your sense of smell. If you don't know what good code looks like and know what bad code smells like, then why would you ever improve it?

Write code to be read

I once heard Joel Spolsky say that Stack Exchange optimizes not for the person asking questions but for the person reading the answers. Why? Because there are far more of them than the single person who asks the question: utility is maximized by optimizing for readers, not questioners.

I think you can view code in a similar way. It will be written just once by you and you alone. However it will be read and edited many, many times, by many others. Your code has two functions: the first is its immediate job. The second is to get out of the way of everyone who comes after you, and it should therefore always be optimized for readability and resilience.

Write your code through the eyes of someone who is coming at it fresh in a year's time.

What assumptions have you made, what do your methods actually return, what on earth does that quadruple nested if / else / and not / unless, statement actually select for? Sometimes you'll need more than just good variable naming and you should ring fence it with tests, but do what it takes (and only just what it takes) to make it durable. Great code is code that does its job and that continues to do its job even when git blame returns a who's who of your company payroll.

Write every line to be read through the eyes of a disinterested and time-pressured team mate needing to extend it in a year's time. Remember that that disinterested and pressured team mate may be you.

Weigh features on their lifetime cost, not their implementation cost

New developers want to explore and to play. They love the latest shiniest things. Whether they're no-sequel databases or high concurrency mobile servers, they want to unwrap all the toys as fast as possible, run out of the room to play with them, and leave the mess for the next dev to clear up.

Dogs aren't just for Christmas and features aren't just for the next release.

Features and architecture choices have maintenance costs that affect everything you ever build on top of them. Abstractions leak, and the deeper you bury badly insulated abstractions, the more things will get stained or poisoned when they leak through.

Experimental architecture and shiny features should be embarked on very carefully and only for very good reasons. Build the features you need before the features you want, and be VERY careful about architecture. Save toys for side projects. Every component you invent, every piece of bleeding edge, fast-changing software you incorporate will bleed and break directly into your project. If you don't want to spend the latter stages of the project doing nothing but mopping up blood, then don't use it in the first place.

Or, as a friend once tweeted:

"Stop being a hipster, and just use Postgres." — @tonymillion

Understand the liability AND the leverage of Technical Debt

Technical debt is the code you write which, while sub-optimal, gets you to where you need to go. It's the errors which, while annoying, are still sub-critical. It's the complexity of a single-app architecture when you know that the future lies in service-orientation. It's the twenty-minute cron job which could be refactored to twenty seconds.

The cost of these not only adds up — it compounds. Einstein once said that "there is no force so powerful in the universe as compound interest." Equally there is no force more destructive in a large software project as compounding technical debt. Most of us have seen (or built) these projects. Codebases where even the smallest change takes months of time. Codebases where people have lost the will to write good code and hope only to get in and get back out without bringing the site down.

Technical debt is an awful burden on a project.

Except when it's not.

Like all debt, when used correctly, technical debt can give you tremendous leverage.

Not only that, but technical debt is the best type of debt in the world, because you don't always have to pay it back. When you build out a feature that turns out to be wrong, when you build out a product which turns out not to work, you will drop it and move on. You will also drop every optimization, every test, and every refactoring you ever wrote for that feature. So if you don't absolutely need them; don't write them. This is the time to maximize your leverage, leave gaps, ignore errors, test only what you need to test.

In the early stages of a product or a feature, the likelihood is that what you are building is wrong. You are in an exploratory phase. You will pivot both on product and on technical implementation. This is the time to borrow heavily on technical debt. This is not the time to fix those sporadic errors or to do massive refactorings. This is the time to run through with guns blazing and keep firing till you burst out the other side.

When that happens though, when you're sure that you're in the right place and out the other side, then it's time to tidy up and to strengthen your position. Get things in good enough shape to keep on rolling, and repay enough of the debt to get you on to the next stage.

Technical debt is (like so many other things in a startup) a game of leapfrog. Your initial code is scouting code. It should move you forward fast, illuminate the problem and the solution, and give you just enough space to build camp. The longer you stay, the more of the system that camp has to support, the bigger and stronger you build it. If you're only ever staying for a week, though, don't burn time laying down infrastructure to support a decade.

Check and re-check your code. Your problems are yours to fix

Engineers who "throw code over the fence" are awful engineers. You should make sure your code works. It's not the testers' job and it's not your teammates' job. It's your job. Lazily written code slows you down, increases cycle times, releases bugs, and pisses everyone off.

If you constantly commit code that breaks things then you are a constant tax on the rest of your team.

Don't kid yourself that you're anything less than a burden and get it fixed.

Do actual work for at least (only) four hours every day

For all the talk about self optimization, focus, and life hacking that goes on amongst developers, the simple truth is that you don't need to do that much work to be effective. What really matters is that you do it consistently. Do proper work for at least four solid hours each day, every day, and you will be one the best contributing members of your team.

However, doing four hours of work every day is harder than it seems.

Proper work is work that includes no email, no Hacker News, no meetings, no dicking around. It means staying focused for at least 45 minutes at a time. Four hours of work a day means that one day lost in meetings or on long lunches and foosball breaks means you have to do eight hours the next one. Believe me, eight hours of solid work is almost impossible. Four hours a day on average also means you should be aiming for five or six in order to prep for the day when you only get two.

However it also means you can be a huge contributor to your team while still having a fully rounded life. It means that you don't need to post that self-indulgent "I'm burning out, please help me" post on HN. It means that by simply being consistent, you can be valued and respected.

Software teams don't slow down because people work four pure hours a day rather than seven (which is insanely hard to do consistently by the way). They slow down because people spend weeks with no direction, or because the louder and emptier vessels dedicate their paid time to discussing Google vs. Facebook's acquisition strategies over endless extended coffee breaks.

Just work. Doesn't matter how incremental or banal your progress seems....

Do four pure hours of work each day, every day and you'll be one of the best people on your team.

Write up the things you've done and share them with the team

However you document things, whether it's through a mailing list like Copyin, a wiki, or even just inline documentation in the code, you should take the time to explain your architectural approach and learnings to the rest of the team.

Have a tough time getting a fresh install of Postgres or ImageMagik to work? If you found it hard, the rest of your team will probably also find it hard, so take a moment to throw down a few paragraphs telling them what you did and saving them your hassles.

One of the worst parts of programming is losing whole days to battling bugs or installation issues. If you take the time to document and distribute the way you found, you could buy back five-times your wasted time by forearming your colleagues.

Understand and appreciate the exquisite balance between too much testing and too little

Testing is a powerful tool. It allows you to set a baseline for the reliability of your releases and makes you less fearful to make them. The less fearful you are to release, the more you do so and the faster you improve.

However it's also an overhead. Tests take time to write, time to run, and even more time to maintain.

> *Think of testing like armour. The more of it you wear the harder it is to hurt you but the harder it is to fight too.*

You become too heavy to move, too encumbered to flex your limbs, immobile. Too little testing, and the first skid across a concrete floor is going to cut you open and leave you bleeding.

There is no intuitive answer to what the right amount of testing is. Some projects require more testing than others, and testing is a whole new piece of expertise you need to learn in and of itself.

Take the time to understand what really needs tests and how to write good tests. Take the time to see when tests add value and what the least you need from them really is. Don't be afraid to test, but don't be afraid not to test either. The right point is a balance; take time to explore where the equilibrium lies.

Make your team better

This is different from the other points in that it's not something you can action so much as an indicator of whether your other actions are working.

Does your presence make your team better or worse? Does the quality of your code, your documentation, and your technical skills help those around you? Do you inspire and encourage your teammates to become better developers? Or are you the one who causes the bugs, argues during stand-ups, and wastes hours discussing irrelevant architectural nuances because it helps cover the fact that you've done no actual work?

You should make your team better. There should always be at least one or two ways in which you make those people around you better and through which you strengthen them. However, be aware that being "smart" alone is probably the least valuable and arguably most destructive dimension you can choose. In fact, if your chosen dimension doesn't actually make you tired, it's probably not a valid one.

It's not who you are on the inside that defines you

There is one humbly brilliant line in Batman Begins which has always stayed with me. At some point in the film where he's fooling around and acting up as a billionaire playboy, Christian Bale implores Katie Holmes to believe that he's still a great guy on the inside. She answers simply: "it's not who you are underneath, it's what you do that defines you."

Your contribution as a developer is defined not by the abstraction of how smart you are or how much you know. It's not defined by the acronyms on your resume, the companies you've worked at, or which college you went to. They hint at what you're capable of, but who you are is defined by what you do and how that changes the projects and the people around you.

If you want to be good, apply yourself. ■

Peter Nixey is a Rails developer and entrepreneur. After starting his career in computer vision at Oxford he then pivoted hard to Consumer Web and on getting investment from YCombinator, took his first company to San Francisco where it was later acquired. He has since built and sold software across the full stack and is now the CTO of Brojure which lets you easily create online e-brochures.

Scalable Program Architectures

By GABRIEL GONZALEZ

HASKELL DESIGN PATTERNS differ from mainstream design patterns in one important way:

- **Conventional architecture:** Combine several components of type `A` together to generate a "network" or "topology" of type `B`.
- **Haskell architecture:** Combine several components of type `A` to generate a new component of the same type `A`, indistinguishable in character from its substituent parts.

This distinction affects how the two architectural styles evolve as code bases grow.

The conventional architecture requires layering abstraction on top of abstraction:

> *Oh no, these* `B`*s are not connectable, so let's make a network of* `B`*s and call that a* `C`.
>
> *Well, I want to assemble several* `C`*s, so let's make a network of* `C`*s and call that a* `D`....

Wash, rinse, and repeat until you have an unmanageable tower of abstractions.

With a Haskell-style architecture, you don't need to keep layering on abstractions to preserve combinability. When you combine things together the result is still itself combinable. You don't distinguish between components and networks of components.

In fact, this principle should be familiar to anybody who knows basic arithmetic. When you combine a bunch of numbers together you get back a number:

```
3 + 4 + 9 = 16
```

Zero or more numbers go in and exactly one number comes out. The resulting number is itself combinable. You don't have to learn about "web"s of numbers or "web"s of "web"s of numbers.

If elementary school children can master this principle, then perhaps we can, too. How can we make programming more like addition?

Well, addition is simple because we have `(+)` and 0. `(+)` ensures that we can always convert more than one number into exactly number:

```
(+) :: Int -> Int -> Int
```

... and 0 ensures that we can always convert less than one number into exactly one number by providing a suitable default:

```
0 :: Int
```

This will look familiar to Haskell programmers: these type signatures resemble the methods of the `Monoid` type class:

```
class Monoid m where
    -- `mappend` is analogous to `(+)`
    mappend :: m -> m -> m

    -- `mempty` is analogous to `0`
    mempty  :: m
```

In other words, the `Monoid` type class is the canonical example of this Haskell architectural style. We use `mappend` and `mempty` to combine 0 or more ms into exactly 1 m. The resulting m is still combinable.

Not every Haskell abstraction implements `Monoid`, nor do they have to, because category theory takes this basic `Monoid` idea and generalizes it to more powerful domains. Each generalization retains the same basic principle of preserving combinability.

For example, a `Category` is just a typed monoid, where not all combinations type-check:

```
class Category cat where
    -- `(.)` is analogous to `(+)`
    (.) :: cat b c -> cat a b -> cat a c

    -- `id` is analogous to `0`
    id  :: cat a a
```

... a `Monad` is like a monoid where we combine functors "vertically":

```
-- Slightly modified from the original type class
class Functor m => Monad m where
    -- `join` is analogous to `(+)`
    join :: m (m a) -> m a

    -- `return` is analogous to `0`
    return :: a -> m a
```

... and an `Applicative` is like a monoid where we combine functors "horizontally":

```
-- Greatly modified, but equivalent to, the
original type class
class Functor f => Applicative f where
    -- `mult` is is analogous to `(+)`
    mult :: f a -> f b -> f (a, b)

    -- `unit` is analogous to `0`
    unit :: f ()
```

Category theory is full of generalized patterns like these, all of which try to preserve that basic intuition we had for addition. We convert more than one thing into exactly one thing using something that resembles addition, and we convert less than one thing into exactly one thing using something that resembles zero. Once you learn to think in terms of these patterns, programming becomes as simple as basic arithmetic: combinable components go in and exactly one combinable component comes out.

These abstractions scale limitlessly because they always preserve combinability, therefore we never need to layer further abstractions on top. This is one reason why you should learn Haskell: you learn to how to build flat architectures. ■

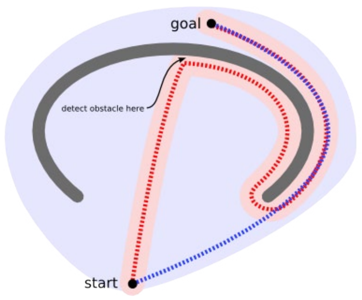
Gabriel Gonzalez builds search tools for biology and designs stream computing and analytics software. He currently works at UCSF where he is completing his PhD in biochemistry and biophysics. He blogs about his work on *haskellforall.com* and you can reach him at *gabriel439@gmail.com*

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

Pathfinding with A*

By AMIT PATEL

MOVEMENT FOR A single object seems easy. Pathfinding is complex. Why bother with pathfinding? Consider the following situation:

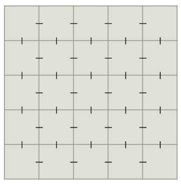


The unit is initially at the bottom of the map and wants to get to the top. There is nothing in the area it scans (shown in pink) to indicate that the unit should not move up, so it continues on its way. Near the top, it detects an obstacle and changes direction. It then finds its way around the "U"-shaped obstacle, following the red path. In contrast, a pathfinder would have scanned a larger area (shown in light blue), but found a shorter path (blue), never sending the unit into the concave shaped obstacle.

Pathfinders let you plan ahead rather than waiting until the last moment to discover there's a problem. There's a tradeoff between planning with pathfinders and reacting with movement algorithms. Planning generally is slower but gives better results; movement is generally faster but can get stuck. If the game world is changing often, planning ahead is less valuable. I recommend using both: pathfinding for big picture, slow-changing obstacles, and long paths; and movement for local area, fast-changing, and short paths.

How A* Works

The pathfinding algorithms from computer science textbooks work on graphs in the mathematical sense — a set of vertices with edges connecting them. A tiled game map can be considered a graph with each tile being a vertex and edges drawn between tiles that are adjacent to each other:

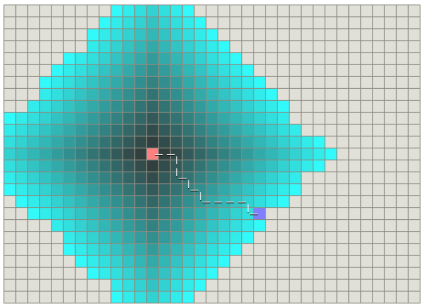


For now, I will assume that we're using two-dimensional grids. Later on, I'll discuss how to build other kinds of graphs out of your game world.

Most pathfinding algorithms from AI or algorithms research are designed for arbitrary graphs rather than grid-based games. We'd like to find something that can take advantage of the nature of a game map. There are some things we consider common sense, but that algorithms don't understand. We know something about distances: in general, as two things get farther apart, it will take longer to move from one to the other, assuming there are no wormholes. We know something about directions: if your destination is to the east, the best path is more likely to be found by walking to the east than by walking to the west. On grids, we know something about symmetry: most of the time, moving north then east is the same as moving east then north. This additional information can help us make pathfinding algorithms run faster.

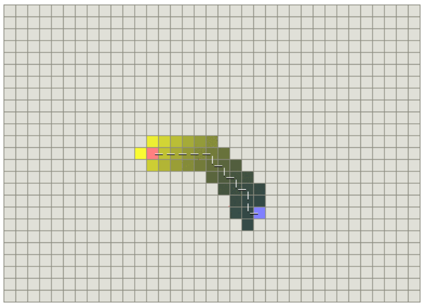
Dijkstra's Algorithm and Best-First-Search

Dijkstra's algorithm works by visiting vertices in the graph starting with the object's starting point. It then repeatedly examines the closest not-yet-examined vertex, adding its vertices to the set of vertices to be examined. It expands outwards from the starting point until it reaches the goal. Dijkstra's algorithm is guaranteed to find a shortest path from the starting point to the goal, as long as none of the edges have a negative cost. In the following diagram, the pink square is the starting point, the blue square is the goal, and the teal areas show what areas Dijkstra's algorithm has scanned. The lightest teal areas are those farthest from the starting point, and thus form the "frontier" of exploration:

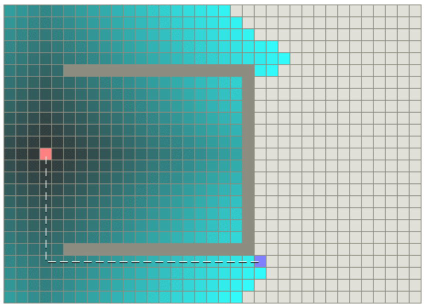


The Greedy Best-First-Search algorithm works in a similar way, except that it has some estimate (called a *heuristic*) of how far from the goal any vertex is. Instead of selecting the vertex closest to the starting point, it selects the vertex closest to the goal. Greedy Best-First-Search is not guaranteed to find a shortest path. However, it runs much quicker than Dijkstra's algorithm because it uses the heuristic function to guide its way towards the goal very quickly. For example, if the goal is to the south of the starting position, Greedy Best-First-Search will tend to focus on paths that lead southwards.

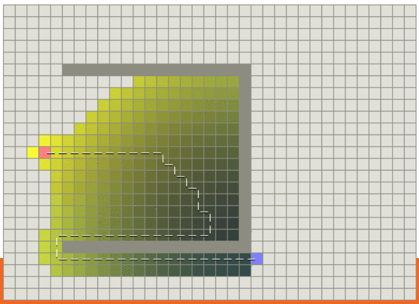
In the following diagram, yellow represents those nodes with a high heuristic value (high cost to get to the goal) and black represents nodes with a low heuristic value (low cost to get to the goal). It shows that Greedy Best-First-Search can find paths very quickly compared to Dijkstra's algorithm:



However, both of these examples illustrate the simplest case — when the map has no obstacles, and the shortest path really is a straight line. Let's consider the concave obstacle as described in the previous section. Dijkstra's algorithm works harder but is guaranteed to find a shortest path:



Greedy Best-First-Search on the other hand does less work but its path is clearly not as good:



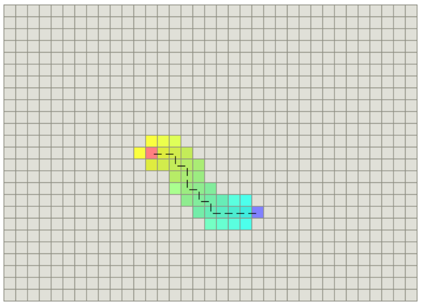
The trouble is that Greedy Best-First-Search is "greedy" and tries to move towards the goal even if it's not the right path. Since it only considers the cost to get to the goal and ignores the cost of the path so far, it keeps going even if the path it's on has become really long.

Wouldn't it be nice to combine the best of both? A* was developed in 1968 to combine heuristic approaches like Greedy Best-First-Search and formal approaches like Dijsktra's algorithm. It's a little unusual in that heuristic approaches usually give you an approximate way to solve problems without guaranteeing that you get the best answer. However, A* is built on top of the heuristic, and although the heuristic itself does not give you a guarantee, A* can guarantee a shortest path.

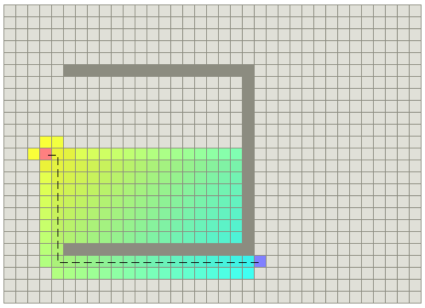
The A* Algorithm

I will be focusing on the A* Algorithm [hn.my/astar]. A* is the most popular choice for pathfinding because it's fairly flexible and can be used in a wide range of contexts.

A* is like Dijkstra's algorithm in that it can be used to find a shortest path. A* is like Greedy Best-First-Search in that it can use a heuristic to guide itself. In the simple case, it is as fast as Greedy Best-First-Search:



In the example with a concave obstacle, A* finds a path as good as what Dijkstra's algorithm found:



The secret to its success is that it combines the pieces of information that Dijkstra's algorithm uses (favoring vertices that are close to the starting point) and information that Greedy Best-First-Search uses (favoring vertices that are close to the goal). In the standard terminology used when talking about A*, `g(n)` represents the exact cost of the path from the starting point to any vertex `n`, and `h(n)` represents the heuristic estimated cost from vertex `n` to the goal. In the above diagrams, the yellow `(h)` represents vertices far from the goal and teal `(g)` represents vertices far from the starting point. A* balances the two as it moves from the starting point to the goal. Each time through the main loop, it examines the vertex `n` that has the lowest `f(n) = g(n) + h(n)`.

Heuristic Functions

The heuristic function `h(n)` tells A* an estimate of the minimum cost from any vertex `n` to the goal. A* uses the heuristic to search the graph more quickly.

- At one extreme, if `h(n)` is 0, then only `g(n)` plays a role, and A* turns into Dijkstra's algorithm, which is guaranteed to find a shortest path.
- If `h(n)` is always lower than (or equal to) the cost of moving from `n` to the goal, then A* is guaranteed to find a shortest path. The lower `h(n)` is, the more node A* expands, making it slower.
- If `h(n)` is exactly equal to the cost of moving from `n` to the goal, then A* will only follow the best path and never expand anything else, making it very fast. Although you can't make this happen in all cases, you can make it exact in some special cases. It's nice to know that given perfect information, A* will behave perfectly.
- If `h(n)` is sometimes greater than the cost of moving from `n` to the goal, then A* is not guaranteed to find a shortest path, but it can run faster.
- At the other extreme, if `h(n)` is very high relative to `g(n)`, then only `h(n)` plays a role, and A* turns into Greedy Best-First-Search.

On a grid, there are well-known heuristic functions to use. **Use the distance heuristic that matches the allowed movement:**

- On a square grid that allows **4 directions** of movement, use Manhattan distance (L_1).
- On a square grid that allows **8 directions** of movement, use Diagonal distance (L_∞).
- On a square grid that allows **any direction** of movement, you might or might not want Euclidean distance (L_2). If A* is finding paths on the grid but you are allowing movement not on the grid, you may want to consider other representations of the map.
- On a hexagon grid that allows **6 directions** of movement, use Manhattan distance adapted to hexagonal grids.

Do not use Euclidean distance squared.

Performance

The main loop of A* reads from a priority queue, analyzes it, and inserts nodes back into the priority queue. In addition, it tracks which nodes have been visited. To improve performance, consider:

- Can you decrease the size of the graph? This will reduce the number of nodes that are processed, both those on the path and those that don't end up on the final path. Consider navigation meshes instead of grids. Consider hierarchical map representations. [hn.my/hierarchical]
- Can you improve the accuracy of the heuristic? This will reduce the number of nodes that are not on the final path. The closer the heuristic to the actual path length (not the distance), the fewer nodes A* will explore. Consider these heuristics [hn.my/heuristic] for grids. Consider ALT (A*, Landmarks, Triangle Inequality) for graphs in general (including grids).
- Can you make the priority queue faster? Consider other data structures such as binary heaps for your priority queue. Consider processing nodes in batches, as fringe search does. Consider approximate sorting.

- Can you make the heuristic faster? The heuristic function is called for every open node. Consider caching its result. Consider in-lining the call to it.

For grid maps, see these suggestions. [hn.my/grids]

Non-grid Maps

Through most of this document I've assumed that A* was being used on a grid of some sort, where the "nodes" given to A* were grid locations and the "edges" were directions you could travel from a grid location. However, A* was designed to work with arbitrary graphs, not only grids. There are a variety of map representations that can be used with A*.

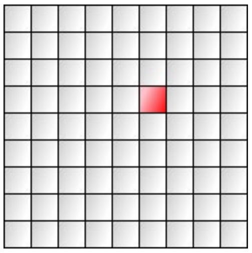
The map representation can make a huge difference in the performance and path quality.

Pathfinding algorithms tend to be worse than *linear*: if you double the distance needed to travel, it takes *more* than twice as long to find the path. You can think of pathfinding as searching some area like a circle — when the circle's diameter doubles, it has *four* times the area. In general, the fewer nodes in your map representation, the faster A* will be. Also, the more closely your nodes match the positions that units will move to, the better your path quality will be.

The map representation used for pathfinding does not have to be the same as the representation used for other things in the game. However, using the same representation is a good starting point, until you find that you need better paths or more performance.

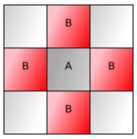
Grids

A grid map uses a uniform subdivision of the world into small regular shapes sometimes called "tiles." Common grids in use are square, triangular, and hexagonal. Grids are simple and easy to understand, and many games use them for world representation; thus, I have focused on them in this document.



I used grids for BlobCity [hn.my/blobcity] because the movement costs were different in each grid location. If your movement costs are uniform across large areas of space (as in the examples I've used in this document), then using grids can be quite wasteful. There's no need to have A* move one step at a time when it can just skip across the large area to the other side. Pathfinding on a grid also yields a path on grids, which can be post-processed to remove the jagged movement. However, if your units aren't constrained to move on a grid, or if your world doesn't even use grids, then pathfinding on a grid may not be the best choice.

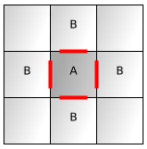
Tile Movement



Even within grids, you have a choice of tiles, edges, and vertices for movement. Tiles are the default choice, especially for games in which units only move to the center of a tile. In this diagram, the unit at A can move to any of the spots marked B. You may also wish to allow diagonal movement, with the same or higher movement cost.

If you're using grids for pathfinding, your units are not constrained to grids, *and* movement costs are uniform, you may want to straighten the paths by moving in a straight line from one node to a node far ahead when there are no obstacles between the two.

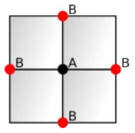
Edge Movement



If your units can move anywhere within a grid space, or if the tiles are large, think about whether edges or vertices would be a better choice for your application.

A unit usually enters a tile at one of the edges (often in the middle) and exits the tile at another edge. With pathfinding on tiles, the unit moves to the center of the tile, but with pathfinding on edges, the unit will move directly from one edge to the other. I wrote a Java applet demo of road drawing [hn.my/roads] between edges that might help illustrate how edges can be used.

Vertex Movement

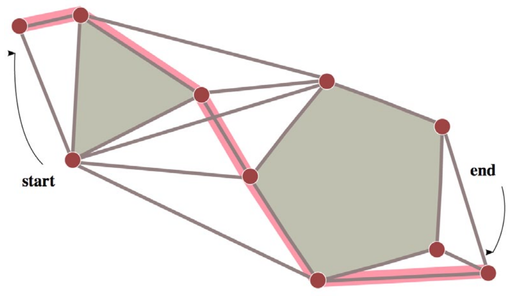


Obstacles in a grid system typically have their corners at vertices. The shortest path around an obstacle will be to go around the corners. With pathfinding on vertices, the unit moves from corner to corner. This produces the least wasted movement, but paths need to be adjusted to account for the size of the unit.

Polygonal Maps

The most common alternative to grids is to use a polygonal representation. If the movement cost across large areas is uniform, and if your units can move in straight lines instead of following a grid, you may want to use a non-grid representation. You can use a non-grid graph for pathfinding even if your game uses a grid for other things.

Here's a simple example of one kind of polygonal map representation. In this example, the unit needs to move around two obstacles:



Imagine how your unit will move in this map. The shortest path will be between corners of the obstacles. So we choose those corners (red circles) as the key "navigation points" to tell A* about; these can be computed once per map change. If your obstacles are aligned on a grid, the navigation points will be aligned with the vertices of the grid. In addition, the start and end points for pathfinding need to be in the graph; these are added once per call to A*.

In addition to the navigation points, A* needs to know which points are connected. The simple algorithm is to build a visibility graph: pairs of points that can be seen from each other. The simple algorithm may be fine for your needs, especially if the map doesn't change during gameplay, but you may need a more sophisticated algorithm if the simple one is too slow. In addition, since we have added the start and end navigation points to the graph, we check line of sight from those to existing vertices and each other, and add edges where needed.

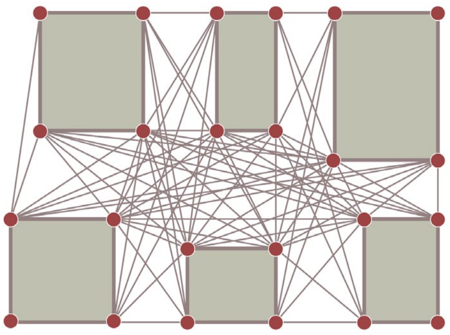
The third piece of information A* needs is travel times between the points. That will be Manhattan distance or diagonal grid distance if your units move on a grid, or straight line distance if they can move directly between the navigation points.

A* will then consider paths from navigation point to navigation point. The pink line is one such path. This is much faster than looking for paths from grid point to grid point, when you have only a few navigation points, instead of lots of grid locations. When there are no obstacles in the way, A* will do very well — the start point and end point will be connected by an edge, and A* will find that path immediately, without expanding any other navigation points. Even when there are obstacles to consider, A* will jump from corner to corner until it finds the best path, which will still be much faster than looking for a path from a grid location to another.

Wikipedia has more about visibility graphs [hn.my/vgraph] from the robotics literature.

Managing Complexity

The above example was rather simple and the graph is reasonable. In some maps with lots of open areas or long corridors, a problem with visibility graphs becomes apparent. A major disadvantage of connecting every pair of obstacle corners is that if there are N corners (vertices), you have up to N2 edges. This example demonstrates the problem:



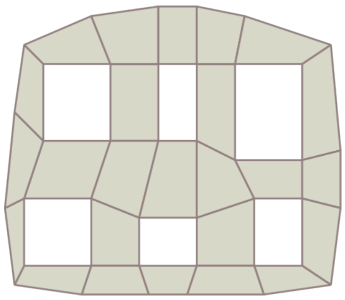
These extra edges primarily affect memory usage. Compared to grids, these edges provide "shortcuts" that greatly speed up pathfinding. There are algorithms for simplifying the graph by removing redundant edges. However, even after removing redundancies, there will still be a large number of edges.

Another disadvantage of the visibility graphs is that we have to add start/end nodes along with their new edges to the graph for every invocation of A*, and then remove them after we find a path. The nodes are easy to add but adding edges requires line of sight from the new nodes to all existing nodes, and that can be slow in large maps. One optimization is to only look at nearby nodes. Another option is to use a *reduced visibility graph* that removes the edges that aren't tangent to both vertices (these will never be in the shortest path).

Navigation Meshes

Instead of representing the *obstacles* with polygons, we can represent the *walkable* areas with non-overlapping polygons, also called a navigation mesh. The walkable areas can have additional information attached to them (such as "requires swimming" or "movement cost 2"). Obstacles don't need to be stored in this representation.

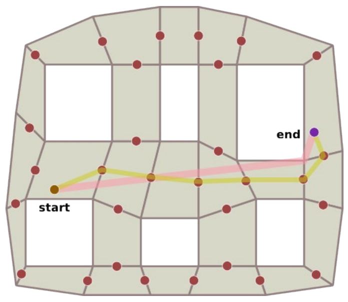
The previous example becomes this:



We can then treat this much like we treat a grid. As with a grid, we have a choice of using polygon centers, edges, or vertices as navigation points.

Polygon Movement

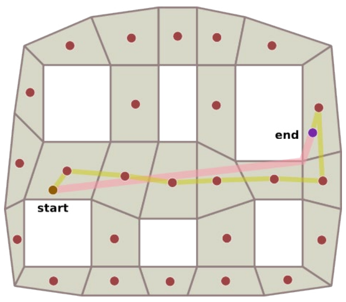
As with grids, the center of each polygon provides a reasonable set of nodes for the pathfinding graph. In addition, we have to add the start and end nodes, along with an edge to the center of the polygon we're in. In this example, the yellow path is what we'd find using a pathfinder through the polygon centers, and the pink path is the ideal path.



The visibility graph representation would produce the pink path, which is ideal. Using a navigation mesh makes the map manageable but the path quality suffers. We can make the path look better by smoothing it.

Polygon edge movement

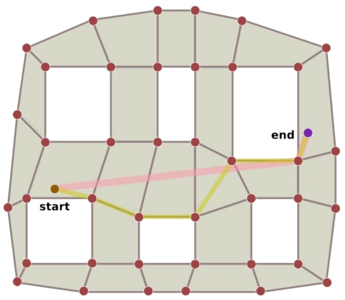
Moving to the center of the polygon is usually unnecessary. Instead, we can move through the edges between adjacent polygons. In this example, I picked the center of each edge. The yellow path is what we'd find with a pathfinder through the edge centers, and it compares pretty well to the ideal pink path.



You can pick more points along the edge to produce a better path, at increased cost.

Polygon Vertex Movement

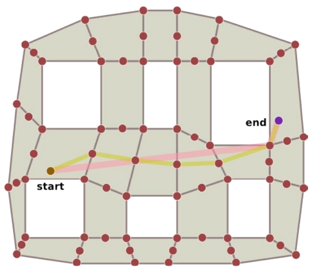
The shortest way around an obstacle is to go around the corner. This is why we used corners for the visibility graph representation. We can use vertices with navigation meshes:



There's only one obstacle in the way in this example. When we need to go around the obstacle, the yellow path goes through a vertex, just as the pink (ideal) path does. However, whereas the visibility graph approach would have a straight line from the start point to the corner of the obstacle, the navigation mesh adds some more steps. These steps typically should not go through vertices, so the path looks unnatural, with "wall hugging" behavior.

Hybrid Movement

There aren't any restrictions on what parts of each polygon can be made into navigation points for pathfinding. You can add multiple points along an edge, and the vertices are good points too. Polygon centers are rarely useful. Here's a hybrid scheme that uses both the edge centers and vertices:



Note that to get around the obstacle, the path goes through a vertex, but elsewhere, it can go through edge centers.

Path Smoothing

Path smoothing is fairly easy with the resulting paths, as long as the movement costs are constant. The algorithm is simple: if there's line of sight from the navigation point *i* to point *i+2*, remove point *i+1*. Repeat this until there is no line of sight between adjacent points in the path.

What will be left is only the navigation points that go around the corners of obstacles. These are vertices of the navigation mesh. If you use path smoothing, there's no need to use edge or polygon centers as navigation points; use only the vertices.

In the above examples, path smoothing would turn the yellow path into the pink one. However, the pathfinder has no knowledge of these shorter paths, so its decisions won't be optimal. Shortening the path found in an approximate map representation (navigation meshes) will not always produce paths that are as good as those found in a more exact representation (visibility graphs).

Graph Format Recommendations

Start by pathfinding on the game world representation you already use. If that's not satisfactory, consider transforming the game world into a different representation for pathfinding.

In many grid games, there are large areas of maps that have uniform movement costs. A* doesn't "know" this, and wastes effort exploring them. Create a simpler graph (navigation mesh, visibility graph, or hierarchical representation of the grid map), or use a variant of A* optimized for grid maps. The visibility graph representation produces the best paths when movement costs are constant, and allows A* to run rather quickly, but can use lots of memory for edges. Grids allow for fine variation in movement costs (terrain, slope, penalties for dangerous areas, etc.), use very little memory for edges, but use lots of memory for nodes, and pathfinding can be slow. Navigation meshes are in between. They work well when movement costs are constant in a larger area, allow for some variation in movement costs, and produce reasonable paths. The paths are not always as short as with visibility graph representation, but they are usually reasonable. Hierarchical maps use multiple levels of representation to handle both coarse paths over long distances and detailed paths over short distances. ■

Amit Patel explores visual explanations of math and computer science topics at *redblobgames.com*. He has a wide range of interests and previously worked on programming languages, scientific equipment, data analysis, visualization, geological exploration, simulation of complex systems, economic modeling, maps, trend analysis, artificial intelligence, and web software.

Reprinted with permission of the original author.
First appeared in *hn.my/pathfinding* (theory.stanford.edu/~amitp)

You Don't Read Code, You Explore It

By JAMES HAGUE

I USED TO STUDY the program listings in magazines like Dr. Dobb's, back when they printed the source code to substantial programs. While I learned a few isolated tricks and techniques, I never felt like I was able to comprehend the entirety of how the code worked, even after putting in significant effort.

It wasn't anything like sitting down and reading a book for enjoyment; it took work. I marked up the listings and kept notes as I went. I re-read sections multiple times, uncovering missed details. But it was easy to build up incorrect assumptions in my head, and without any way of proving them right or wrong I'd keep seeing what I wanted to instead of the true purpose of one particular section. Even if the code was readable in the software engineering sense, boundary cases and implicit knowledge lived between the lines. I'd understand 90% of this function and 90% of that function and all those extra ten percents would keep accumulating until I was fooling myself if I thought I had the true meaning in my grasp.

That experience made me realize that read isn't a good verb to apply to a program.

It's fine for hunting down particular details ("let's see how many buffers are allocated when a file is loaded"), but not for understanding the architecture and flow of a nontrivial code base.

I've worked through tutorials in the J language [jsoftware.com] — called "labs" in the J world — where the material would have been opaque and frustrating had it not been interactive. The presentation style was unnervingly minimal: here's a concept with some sentences of high-level explanation, and here are some lines of code that demonstrate it. Through experimentation and trial and error, and simply because I typed new statements myself, I learned about the topic at hand.

Of particular note are Ken Iverson's interactive texts on what sound like dry, mathematical subjects, but they take on new life when presented in exploratory snippets. That's even though they are reliant on J, the most mind-melting and nothing-at-all-like-C language in existence.

I think that's the only way to truly understand arbitrary source code. To load it up, to experiment, to interactively see how weird cases are handled, then keep expanding that knowledge until it encompasses the entire program. I know, that's harder to do with C++ than with Erlang and Haskell (and more specifically, it's harder to do with languages where functions can have wide-ranging side effects that can change the state of the system in hidden ways), and that's part of why interactive, mostly-functional languages can be more pleasant than C++ or Java. ■

James Hague has been Design Director for Red Faction: Guerrilla, editor of "Halcyon Days: Interviews with Classic Computer and Video Game Programmers," co-founder of an indie game studio, and a published photographer. He started his blog "Programming in the 21st Century," in 2007.

2048, Success and Me

By GABRIELE CIRULLI

In March, I built a game called 2048 [git.io/2048] just for fun, and released it as open-source software on GitHub [hn.my/gh2048]. Over the course of the following weeks it unexpectedly became a worldwide hit, and it has been played by more than 23 million people.

This period has been one of the most exciting of my life, as well as one of the most stressful. Knowing that millions of people have played and enjoyed something you've built can be a great feeling. For many (including me), it's what gives you the motivation to keep coming up with new creations. At the same time, when something you made becomes known worldwide you have to face a whole new set of challenges. The attention you get and the things people come to expect of you can become overwhelming if you've never had to handle them.

In this article, I'll share what this experience has been like for me and how I dealt with it, both on a personal and professional level. I will also explain the path that led me to changing my mind on building a mobile version of the game.

It's a long read, but I hope that this article will provide some meaningful insights and hopefully help those who might be facing similar issues.

How it all started

I built 2048 in a weekend, just for fun. I had become addicted to two other games, called 1024! [hn.my/1024] and 2048 [saming.fr/p/2048]. I loved playing both, and I wanted to create my own version with a different visual style and quicker animations, just to see if I could. At that time, I did not know about Threes, [asherv.com/threes] the game from which all the others (including 2048) originated.

Asher Vollmer and Greg Wohlwend, its creators, have poured a huge amount of time and effort into it. They've recently expressed their frustration [hn.my/threemails] over the popularity that the clones of their game experienced. I understand what they must have felt like, and I have a huge appreciation of the amount of work and love they put into building Threes. 2048 owes its existence to it.

While building 2048, I decided that I should just put it on GitHub and be done with it. I didn't feel good about keeping it private, since it was heavily based off of someone else's work.

Once I was done with the game, I published it on GitHub Pages and posted it on Designer News, simply interested in getting feedback over the visuals.

The explosion

The following day, I received a message from a friend telling me to have a look at the front page of Hacker News. Someone had posted 2048 there and it was at the #1 position. Google Analytics reported thousands of people on the site. I just couldn't believe what was happening.

Although it just looked like a sudden spike in interest, one which would fade away quickly, I spent the entire day looking at the stats. Seeing the counts continuously going up made me excited and a little terrified at the same time.

I was surprised by the amount of positive feedback I was getting in the comments. Everyone was talking about how they just couldn't stop playing this game, even at the expense of their productivity.

The following days

I thought the interest in 2048 would fade away soon, but it didn't stop even after a few days. In fact, it just kept getting larger. At some point, 2048 had gone from being a popular topic amongst HN readers (it became the third most up-voted post in the history of the site) to being talked about on Twitter, Facebook, and even offline. Seeing it turn into a worldwide phenomenon felt a bit unsettling.

At the same time, my inbox had started growing with emails from people interested in the game, as well as developers asking for authorization to port the app to mobile to profit off of it.

The first problem I faced was figuring out what I should do about 2048 and how I should respond to those emails. Although 2048 was just a small side project for me, and I had no particular expectations about it, the people around me were suggesting that should I jump at the opportunity to make money out of it.

Personally, I didn't feel comfortable about the idea of profiting off of the concept, since 2048 was mostly based on other games.

What also caused me a considerable amount of distress was knowing that, in order to focus on 2048, I'd have to give up on all my other commitments. At the time, I was working on a freelance project, and focusing on 2048 meant I'd have to pause it or end it altogether.

I had to bring the game to mobile, a field I had no experience in, and do it quickly enough to be first. The prospect of doing this scared me because it would be a big jump out of my comfort zone, having no idea of what lied ahead.

Those two factors caused me a lot of distress during those days. I felt as if there was no way out, and every decision I may take would only lead to more trouble.

On one side, I could embrace this opportunity (which felt like a "once in a lifetime" deal) and get a return, at the cost of wronging the people behind the original concepts.

On the other hand, I could just do nothing and go on with my life. I knew I would regret it when, later on, someone would tell me I missed out on my opportunity.

In the end, I convinced myself that I should just do nothing, because I thought that was the only way to end the stress I was experiencing. I decided that the game would remain open-source, and that I wouldn't bring it to mobile.

After making that decision, I immediately started feeling better. That made me think that I had done the right thing, and I wouldn't regret it.

Falling back into the circle

For the next few days, I felt better again.

At that time, the first mobile versions of the game had started appearing. Some of them would not even credit me or the other games they were based on, some would outright impersonate me. After seeing the reaction of the people behind Threes, I thought that not pursuing this myself had been the right choice.

Many of the people around me, however, didn't feel the same. My friends and parents thought that my choice was honorable, but at the same time I was probably throwing away a chance that I would be unlikely to get a second time.

Initially, their opinions didn't phase me. I knew that by choosing this path I had saved myself from the stress I was feeling before, and I considered that far more important than money or popularity.

This feeling didn't last long, however. A few days later, all of the issues I thought I had overcome crumbled back on me much harder than before. I had started to regret "wasting" this opportunity, and I felt as if the people around me were disappointed by my actions. What made me feel even worse was seeing a 2048 app made by someone else get to the top of the leaderboards in the App Store.

Silver linings

I was distraught because of my situation, but I also had reasons to be happy.

Even though some people don't care about taking someone else's work and using it for profit, there are also many creative people in the open-source community who care about improving what's out there. They just want to take something and make it better, or even bring it to entirely new levels.

Seeing the countless derivative versions of 2048 that had appeared made me incredibly happy. I had a lot of fun playing each one of them, and it just felt great to see what others were coming up with.

People poured a lot of creativity into tweaking the game and shaping it into completely new things. Knowing that someone else spent their time on improving something you've built can be elating, especially if what motivates you the most is just making people happy through your work.

Changing my mind

Thanks to the help of my parents and my friends, I realized that the only way to get over this without feeling like I had missed an opportunity would be to embrace it and produce an app. I wouldn't be doing it for profit, though. In fact, that is not what matters to me. All that matters is knowing that I didn't waste a chance, no matter if I'm going to succeed or fail.

What would people think of me, though? In every interview, I said that I wouldn't try to profit from the game for ethical reasons. I thought that if I changed my mind, I'd be seen as a hypocrite, and I really didn't want to be that kind of guy.

The hardest thing about this decision was that I felt it would betray other people's expectations of me. After all, I would be changing my mind and pursuing something I was outspokenly against. It took me a few days, but what eventually led me to accept this was knowing that my change of heart would not be motivated by greed. I chose to do it to save myself from feeling like I missed my chance for the rest of my life.

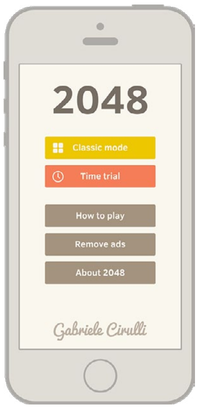
Due to my choices, those who took these issues at heart and appreciated my previous stance will probably feel deceived. That's one of the reasons why I chose to write this article: I wanted to give my perspective over this controversial choice, one which was mandated by extreme conditions.

Hopefully, the context in which this decision was taken will be enough of an explanation for my actions. If not, I hope I at least somehow helped people understand where all of this came from.

Building the application

I still thought that just taking the game as it is and wrapping it in an application would be a bad thing to do. 2048 was not really mine anymore. Instead, it belonged to the countless contributors who believed in it, and I had no right to use it for myself. If I wanted my decision to feel reasonable, I would have to put a considerable amount of work into creating the application.

In the end, it took me a full month to develop the application from scratch and bring it to a state of polish that would motivate its existence.



The main menu of the application.

Being a web developer and having no mobile development skills, I couldn't just start building an app for iOS and Android and expect to come up with anything decent. Because of that, I decided to use Phonegap. Phonegap allows you to build an app using HTML, CSS and JavaScript in the same way you'd build a mobile website.

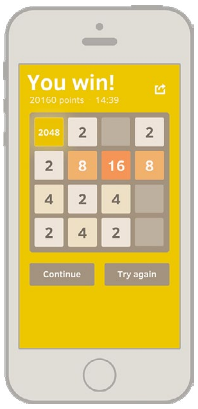
The problem with Phonegap is that if you want to build a native-looking app, you have a long, hard road ahead of you. Fortunately, I was trying to build a game, which meant I wouldn't have to strictly abide to the visual styles of each OS.

I wanted the application to have a menu, because dropping the player into a game when launching the app wouldn't make for a good experience. A menu would also let me introduce new game modes, which would add value to the game.

I wanted the game logic to be generic enough to be allow the inclusion of new game modes just by creating new objects that "hook" into the core game and modify its behaviors as needed.

I ended up writing most of the application's code from scratch. The only part I kept from the open-source version of 2048 is the logic to move the tiles, to keep true to the original experience. Since the app will be closed-source for now (but I might publish it in the future), it wouldn't be fair if it used code that other people contributed.

The codebase turned out to be almost 3 times the size of the web version, with most of the code being new.



The screen you see when you win.

While building the app, I found many ways to improve and streamline the code and the interface. I'd really love to give back to the open-source version of 2048 by porting these back into it. I also want to eventually refactor its code, to make it a better asset for the open-source community.

If you're interested in seeing what I ended up with, you can download the application for iOS [hn.my/i2048] and Android [hn.my/a2048]. I hope you'll like it. ■

Gabriele is from Italy. He graduated from high school a year ago and jumped straight into work afterwards. He began as a freelancer, then after created 2048, he joined a web startup.

BETTER SENDING, BETTER INSIGHTS

ANALYZE, REACT, ENGAGE

NEW

- REST API
- Parse API
- Real-Time event API
- features coming soon!

mailjet™

You **push** it
we **test** it
& **deploy** it

circleci

Get 50% off your first 6 months
circleci.com/?join=hm