My First Six Months of Programming

By Whitaker Blackall

HACKERMONTHLY

Issue 14 July 2011

Curator
Lim Cheng Soon

Contributors
Chad Etzel
Whitaker Blackall
Paul Stamatiou
Diomidis Spinellis
Ruben Berenguel
Fabien Sanglard
Joel Neely
Robin Houston

Proofreader
Emily Griffin
Molly O'Donnell

Printer
MagCloud

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

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

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

Cover Photography: Dana Tesser

Contents

Effort, *flickr.com/photos/krikit/2880756271*

DOOM
id SOFTWARE
PROVIDED BY id FREE OF CHARGE • SUGGESTED RETAIL PRICE $9.00 • id SOFTWARE, ©1993

Startups are Hard

By CHAD ETZEL

STARTUPS ARE HARD. No, startups are *damn hard*.

Contrary to popular belief, there are no clouds of money that float around Silicon Valley and rain on anyone that utters the phrase, "I'm a founder!" Unfortunately, starting a company and raising money is just as hard as ever; it's just that the investors don't have as much leverage as they used to, but they still have a lot.

Most reporting on startups suffers from a terrible case of success bias. Nobody wants to report on a dying startup unless it is to highlight another company that has come along to kill them, but that actually turns into a piece about the better company and not the dying one.

Startups that die rarely talk about it publicly because it is frustrating, embarrassing, and most of the time the people involved want to forget the whole mess and move on rather than sit around talking about the fact that they failed.

Most people don't want to admit that startups are hard, either, because to admit something is hard is to admit that you don't know everything there is to know about a certain topic and to display weakness. If there's one thing you do not want to do as a startup, it's appear weak. Only the strong survive.

But guess what: startups are hard. At times they are soul-crushingly hard. I am not afraid to admit this anymore. I am not afraid to talk openly about it with peers anymore. So, this post serves as a counterpoint to all the recent postings alluding to the fact that anyone can suddenly decide to be a founder and the next week find themselves swimming around in a kiddie-pool full of angel/VC money.

You're Nobody Till Somebody Loves You

In the Valley, you are a Nobody until you are a Somebody. Trying to launch a new product as a Nobody is hard. Trying to get press as a Nobody is very hard, because nobody knows who you are (read: nobody cares who you are) and so they don't care about your product. Press outlets are already so saturated with inbound leads from trusted and credible sources that trying to promote yourself as a Nobody will find your attempts mostly routed to an inbox black hole.

Raising money as a Nobody isn't just hard, it is nearly impossible. There are a few major factors that investors look at when making an investment decision. Two big factors are Traction and Revenue, and they are somewhat orthogonal. It's ok if you have impressive numbers on one axis but not the other, but having both is even better.

One other major factor is Social Proof: that is, are you a Somebody?

Having a track record of past success is a really big deal. This is what allows people to raise money with their name alone. It allows them to walk into a VC office, say "I have a cool idea," and walk out with a fat check. (This is an oversimplification, of course, but it's not too far off.) The fact that the Color team was able to raise $41M is not a surprise to me; in fact that number seems pretty low given the resources they will need to reach their ambitious goals. Being a Somebody affords you lots and lots of advantages (both in getting press and raising money).

So then, the question becomes, how does a Nobody become a Somebody? You need your first big success to become a Somebody. Having your first big hit as a Nobody is the major bootstrap problem that all new founders will face. The trick is to brute force a successful result in the face of nobody caring about who you are. And that, to me, is the hardest part about startups. It demands superior execution and perseverance in the face of the crap-storm you will endure.

I'm still a Nobody in the Valley. Sure, I have a lot of great contacts, a great network of fellow startup folks, and a great resume of past corporate experience, but all that does not a Somebody make. Until I have a track record of at least one past success, I will continue to be a Nobody.

"Jealousy... is a mental cancer." – *B.C. Forbes*

Am I jealous of other companies' success? I would be lying if I said no. I am slightly jealous when I wake up and read another story about some company raising a million dollars for some idea that makes absolutely no sense to me, or seeing an acquisition of a company for a product I did not feel was particularly well executed. I am happy for them because they were able to pull off something that I know firsthand is very hard. At the same time, I am jealous that they were able to figure something out that I honestly haven't been able to yet. The point is, though, I am not just sitting around whining and waiting for my ship to come in. I am actively building my ship from scratch by hand, and one day it will set sail. I am also discovering how to refocus that envious energy into learning from their successful experiences.

Effort, *flickr.com/photos/krikit/2880756271*

“Startups are grueling, and the only way to stay sane is to have some sort of support group, especially outside of work.”

Having a Support Group

Startups are grueling, and the only way to stay sane is to have some sort of support group, especially outside of work. Co-founders and employees are of course wonderful for support, but they are also drinking the same kool-aid. Having some friends outside of work to give different perspectives is very valuable. Since we just moved to the area recently, we haven’t had time to foster many new or deep friendships. So for me, my greatest supporter is my wife.

My wife is a saint. She has supported me every day with encouragement. She is my biggest cheerleader. Some days I am honestly surprised that I don’t wake up and find an empty dresser, a missing suitcase, and a letter on the counter. She has sacrificed every bit as much as I have for no other reason than she believes in me. Some days that is the only motivation moving me forward.

Recently my wife found a job after six months of searching. It’s right up her alley, and she loves it. This has been a real boon both for her morale and for our bank account. Her paycheck now represents a significant portion of our income, and I recognize this fact. Thanks to her job, I am able to reduce my paycheck, and thus give a longer lifespan to my startup.

Someday when this is all over, and assuming the result is that we can afford it, I owe her a month long stay at a 5-star spa.

Sacrifice

Startups demand sacrifice. As a sample size of two, here is a list of things my wife and I have sacrificed in order to go out and chase the American Dream:

- My well-salaried corporate job working on fun and interesting problems.
- A peer group at work that gave me equal amounts of respect, even though I was several years their junior.
- Low cost of living on the East Coast.
- Our 3-story townhouse with a huge kitchen and hardwood floors in the 'burbs traded in for a tiny one-bedroom apartment in San Francisco with insane rent (in addition to still paying the mortgage on our house).
- My awesome sports car that I loved to drive around the mountains.
- Burning through nearly all of our personal savings.
- Health insurance.
- Vacations (read: time-off, since I could still travel around if I really wanted to, but I am never really “off the clock”).
- Monthly contributions to a 401k plan.
- Our great church home in Raleigh.
- My wife’s friends.
- My wife’s job at UNC.
- Nine months living separated from my wife while I went through YC and tried to raise money thereafter.

“Has it all been worth it? If you are expecting me to say “Yes, of course!” you would be wrong.”

- Sold all of our belongings except our clothes so we could move across the country.
- Took on credit card debt for the first time in our lives.
- Left my funk band (happily, they found a replacement and are still jamming!).
- Losing my hair (well ok, this has been happening for a while...).

Has it all been worth it? If you are expecting me to say “Yes, of course!” you would be wrong. The truth is, it hasn’t been worth it at all...yet.

Financially speaking, we are much worse off now than when I took the plunge. Of course the goal is for it to be worth it someday, but it is unclear how long it will ultimately take. In other aspects of my life, it may have been worth it so far, but it is hard to quantify those things. For example, I am now surrounded by a lot of like-minded technical people (which is great!), but overall my quality of life has taken a big hit. How long one can take this is a measure of their true grit, but how long that is for me personally remains to be seen.

Startup Depression

In the case of Notifo and Phrygian Labs, Inc (the parent company of Notifo), we had a terrible time fundraising last fall, even when the investing market was starting to get really hot again. The ultimate reality, though, is that we failed utterly at fundraising. We ended up wasting a lot of time. We had dozens and dozens of intros, which led to about 40 or so meetings. After spending 3 months and hearing “no” 39 times (this includes all the jerks who never bothered to reply with an answer after our meeting and repeated attempts of contacting them) we decided to just give up raising money. We looked around and felt like everyone around us was raising insane rounds with no problem, and here we were just getting stonewalled. The net effect was that it killed our morale dead.

After hearing, “UR DOING IT WRONG!” so many times, it’s hard to think that you’re doing anything right. At that point it’s very hard to soldier on. One of the things I regret now is that we didn’t just carry on in spite of everyone telling us no. We had made a terrible mistake; we had given control to the investors, and they weren’t even giving us money! We let them dictate our path with their negative signaling instead of listening to our guts. We had been told, “...mobile notifications is a hot space, you’ll have no problem raising a ton of money,” so many times that when we failed to do so, it made us feel like total idiots.

Looking back, I wish we had carried on full-bore, but I can also remember after we quit fundraising that I felt like such utter shit that I didn't feel much like doing anything at all, much less writing code. There was literally a week when I would just stay in bed all day so I didn't have to face the world. This was the lowest point in my entire life.

The lesson here is if you are having trouble putting together a round in the first few weeks of actual investor meetings, just say, "screw it," and get back to working ASAP.

On top of all that, in the middle of all that fundraising junk we received an "interesting" acquisition offer from another company. That is another long story for another time, but the end result was that they caught us at our most vulnerable which caused a lot of mental anguish. We basically went back and forth on accepting and rejecting it dozens of times. After seeking advice from Paul Graham about the situation and telling him how much money we had left in the bank he said, "There are starving startups in Africa that don't have that much money. You still have time to make a go of it!" Ultimately, it was not the right offer and we said no.

This whole episode basically set us on a sideways path of non-productivity for the next several months as we flailed around looking for a way to pivot or just reset entirely.

Conclusions

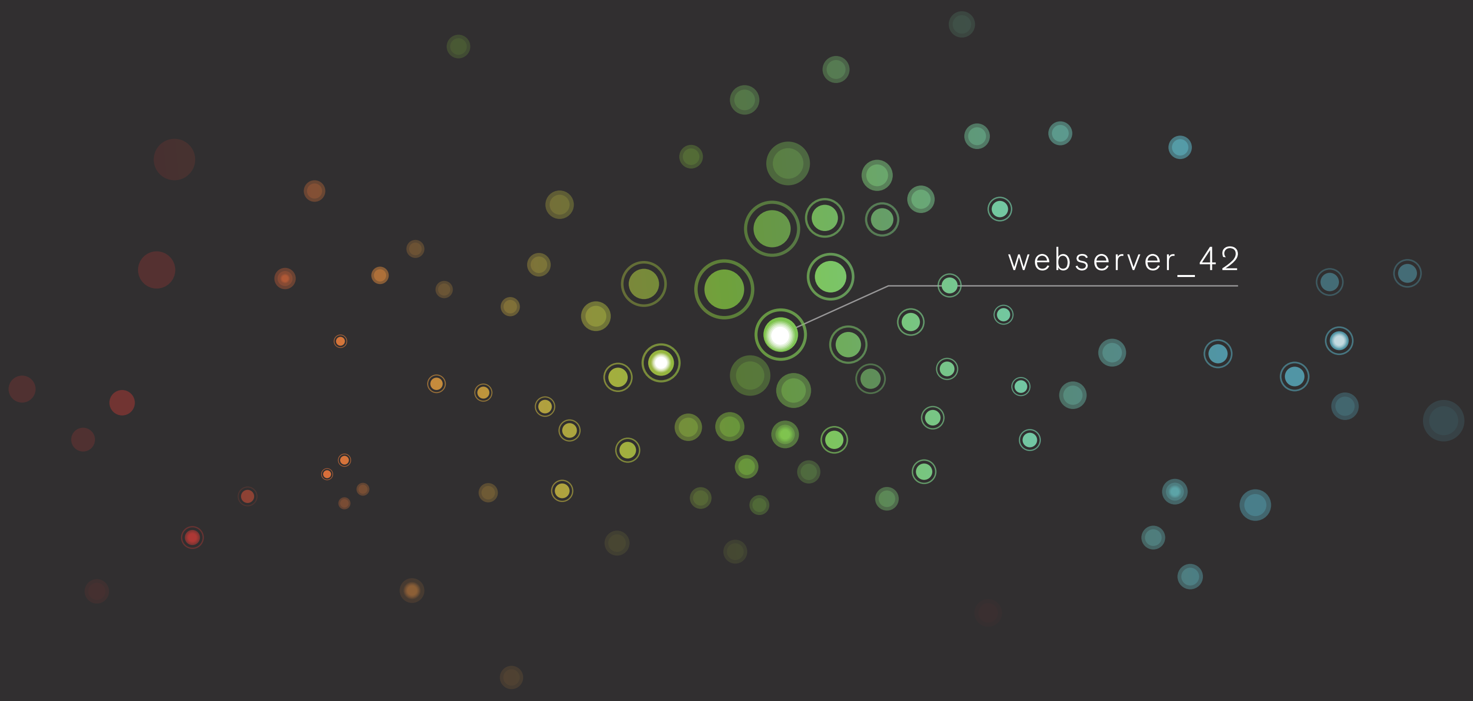
So, what is the point here? The point is that startups are still hard even if nobody is spending time discussing that fact. Admitting this and realizing that all other startups are in your same boat is liberating. Having other friends who are in startups that are willing to openly share their hardships is cathartic. It helps you realize you are not alone when it seems like everyone else is succeeding. Of course it looks like everyone else is succeeding; every company has a duty to seem like they are doing fine from the outside, otherwise everyone (investors, press, and most importantly customers) will lose all confidence in them. Every company has this facade that they are doing really well, and this creates the very convincing illusion that you are the only ones suffering. The reality is, every startup is basically screwed in one way or another. The fact that a large majority of startups fail is a testament to how hard they really are. If startups were easy, everyone would be running them. ■

Chad Etzel is the founder of Notifo.com, a Y Combinator startup. He enjoys writing code for web and native mobile applications. Prior to starting Notifo, Chad worked at Twitter and Cisco. You can find him only at *jazzychad.net*

These are your servers

● ● ●

These are your servers on Cloudkick



Any questions?

cloudkick.com

415.779.5425

support for 8 clouds + dedicated hardware

cloudkick

the best way to manage the cloud

My First 6 Months of Programming

From Man-Rodent to Partyman

By WHITAKER BLACKALL

Photography: Dana Tesser

Almost exactly six months ago, I decided to embark on an adventure. Before October of 2010, I had never programmed a day in my life. Okay, maybe I had a week-long unit on the very, very, very basics in sixth grade, but that is all. I had no idea what to expect. October 19th, 2010 I tweeted: "What have I gotten myself into. I've just embarked on learning programming, having absolutely no experience. Oh man I'm already nervous." That was the beginning of a path that I'm so glad I have taken. I'm not looking back.

Beginnings

I'm going to give you a quick overview of my first few months, because it was pretty boring. I started with Invent with Python [inventwithpython.com], a really awesome book for complete noobs. To show you how beginner it was, here is a quote from the book:

> *"The + sign tells the computer to add the numbers 2 and 2. To subtract numbers use the – sign, and to multiply numbers use an asterisk (*)."*

After I got the very basics down, I made a bunch of random text-based games, like hangman, jotto, and guess-the-number. My first big hit was a game called "The Man-Rodent." It was about a Man-Rodent, whatever the hell that is, who was terrorizing the village. He was hiding, and you had to guess where he was.

This was also when I received my first piece of cherished fan art. I posted the game on Reddit, and Michael Hussinger made this *amazing* cover art for it.

Next, I moved on to my first quasi-graphics based game: Minesweeper. This is when I really started to understand why creating a flexible game engine is really important (yeah, like I created a game engine). The game was still text-based, but it was completely flexible in that I could specify board size and mine amount. This ended up giving it some replay-ability, and it was fun to play around with crazy numbers just to see what happened. Here's that game [hn.my/minesweeper].

Welcome to iOS

At this point, I felt I was ready to move on to actual graphics. I felt at an impasse. I had absolutely no idea where to start. I took a look at SDL but was in way over my head and got quickly discouraged. Since I knew I eventually wanted to make an iPhone app, I asked Matt Rix (Trainyard), who gave me some sage advice. He recommended I start by learning C, then Objective-C, then Cocos2d. I'm happy to say that as of today, I've learned all three of those — the basics anyway.

During the period of learning the basics of C and Obj-C, I came across a lot of frustrations, and I didn't make many games. Many times, I really wondered if I could ever get past the hurdle and figure it out. A few of the things that I pulled my hair out over when I first learned of them: structs, arrays, multi-dimensional arrays, properties, views, view controllers, protocols and delegates, memory management, and many more. And there is still so much out there that is way over my head. But I'm going to keep trucking, because everyone I talk to says the only way to get better is to just keep making games. Being a piano player, I am well aware that practice pays off, so that's what I've been doing (and will keep doing).

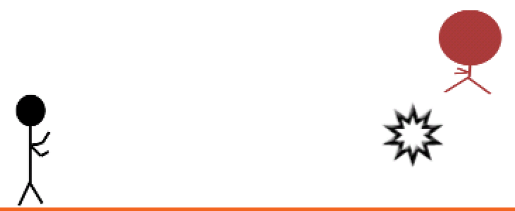
Welcome to Cocos2d

Once I finally got the basics of creating an iPhone app down, I was ready to delve into Cocos2d. I was nervous but excited. Now, I don't know if I'm crazy or stupid, but I have found that the Cocos2d documentation is not very good. All the classes, methods, and properties are listed, but I am constantly struggling to understand things because they are not explained well. An example: every "node" (the main Cocos2d element) has a "(BOOL) is Running" property. It sounds simple enough, but it could easily mean any number of things. All the documentation says about this property is "whether or not the node is running." Uh, okay....Needless to say, I'm still pretty confused about a fair amount of things in Cocos2d.

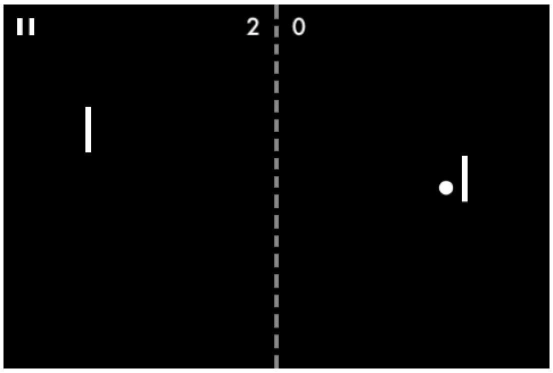
But enough of the insults, I am loving Cocos2d. Once I got a hang of the basics, I started to see how much easier it was to get a simple game up and running in no time. And since I already know how to do music and sound, and also know a little bit of Photoshop, I was really starting to see some pretty cool results. Since I was really confused by Cocos2d at first, I started with a few of Ray Wenderlich's awesome tutorials [hn.my/ray].

A Bunch of Sorta-Games I've Made

Here is my first Cocos2d game, which I created by directly following Ray's tutorial (it even had my own music and stupid sound effects for when an enemy died):

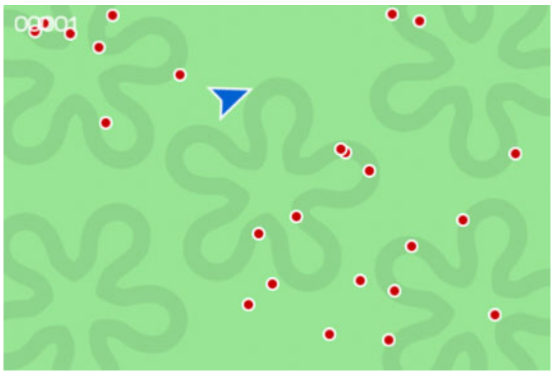


Next, I made a Pong clone. It even had multiplayer:

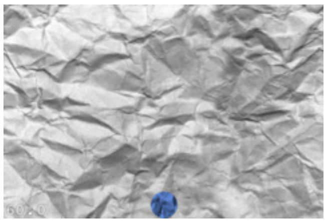


After Pong, I was feeling encouraged and decided to try to make Tetris. Big mistake. I was not ready. I ran into a lot of really annoying errors and bugs, and when I finally got the basic framework down, I tweeted something like: "So proud of myself. I just created a really complex class/subclass system for my Tetris clone." Noel Llopis quickly informed me how bad of an idea that was. So I looked at some tutorials and tried to completely revamp my Tetris clone, but it was still too difficult. I will return and defeat you one day, Tetris!

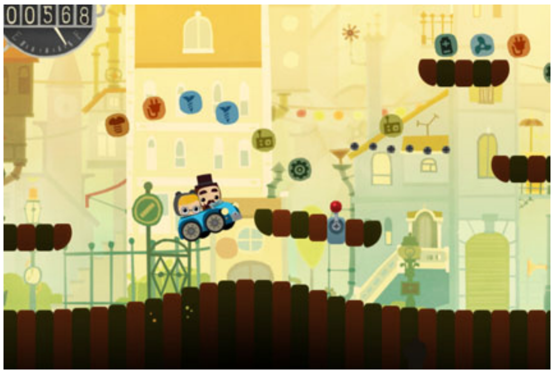
After my failed attempt at Tetris, I made a little Tilt to Live inspired prototype, except in this version you get points for collecting red dots instead of avoiding them. And this one didn't have the music I did for the real game:

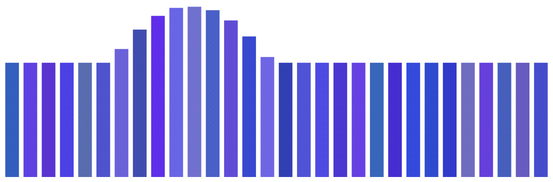


After a lot more learning of Cocos2d, I had a random inspiration to make an endless crumpled paper background. So I crumpled paper and made it an endless background. Then I put a little paper circle in the game that you could roll around using the accelerometer. I was really proud of myself when I figured out how to roll the circle because it involved some math that I hadn't done in a long time (even though it was fairly simple trigonometry). The background was buggy — there were frequently gaps between the images of the repeating background — but it worked.

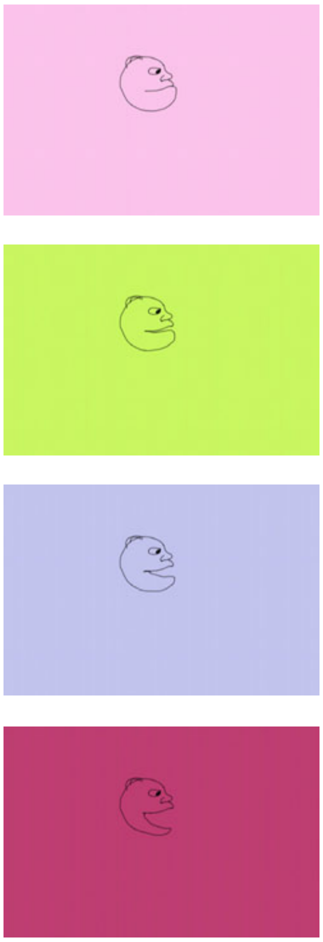


Next, after seeing a trailer for the awesome-looking upcoming game Bumpy Road, I got inspired and wondered if I could recreate an effect similar to its bumpy road element. Needless to say, my graphics were a bit more barren, but you can compare.

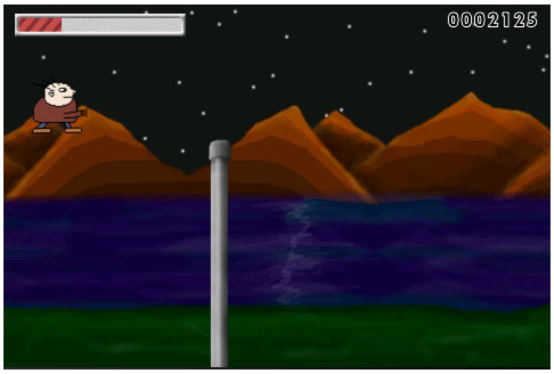

Bumpy Road

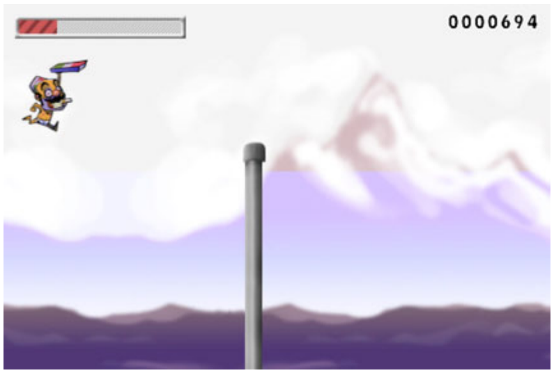

My version

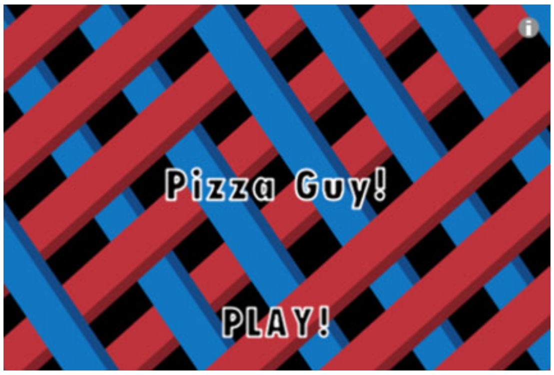
After this, I wanted to try my hand at actually animating something. I had never animated anything before and gave it a go on Photoshop. Man, was it tedious! Even for just a stupid, little line drawing. But what I came up with was someone I like to call PARTYMAN. All he did was open and close his mouth over and over again, each time making a randomly chosen grunt noise. You could also move him around the screen, and the background flashed different colors. And he only grunted and partied when you were actually touching the screen.

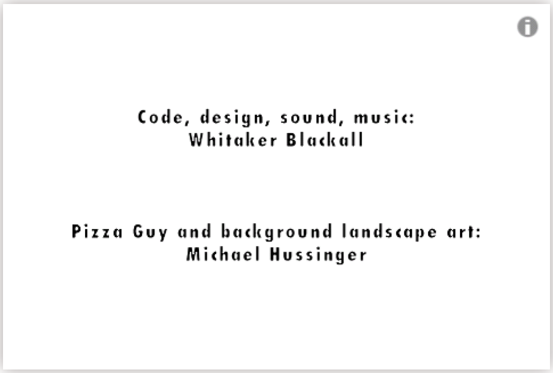


Somehow, PARTYMAN gained a small following. I think he might be next in the long list of icons including Mario, Link, Samus, and others. I actually ended up sending a copy to Jared of Touch Arcade (hey, he asked for it). Then, to my great surprise and utmost joy, Craig Sharpe of Retro Dreamer created my second official piece of fan art! Here is Craig's AWESOME take on PARTYMAN.







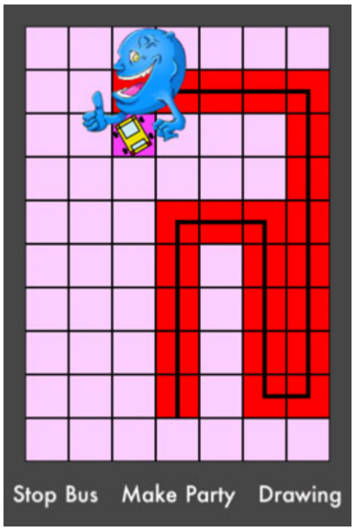

Code, design, sound, music:
Whitaker Blackall

Pizza Guy and background landscape art:
Michael Hussinger

His drawing was so awesome, in fact, that I had to include it in one of my next games. But first, on a whim I started making a game about a guy jumping over poles. The original point was just to see if I could make a parallax background with a simple game mechanic, but I just kept going with it. It ended up having a title screen, score-tracking, replayability, music, sounds, variable jump heights, a jump height progress bar indicator, and even a credits page! It was my most complete game yet. I drew some pretty awful sprites for my game, as you can see above.

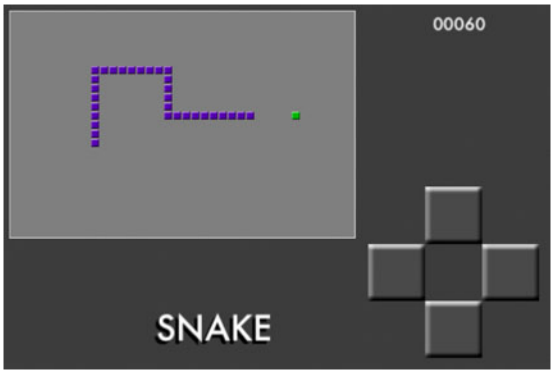
Since my drawings sucked, I asked Michael Hussinger (of "The Man-Rodent is in the Barn" fame) if he wanted to do a few quick doodles for it, and he did! He drew this awesome pizza guy in probably 20 minutes. It would take me hours upon hours to draw something that good, if I even could at all. He also drew a nice mountainous background for my parallax scrolling. Here's how the game looked after his help, along with my title and credits pages.

My next game was inspired by Trainyard. I wanted to see if I could create a grid that I could draw roads on that a vehicle would follow. What better opportunity for a return of PARTYMAN, right? So I called it Partybus. The point of the game was to click a spot on the grid to make a party (a flashing, color-changing spot with music coming out of it), then to draw a road to the party and have PARTYMAN drive there in his Partybus. Needless to say, it was a really easy game. But the cool thing is, when he reached his party destination, the music got louder and more bass, the bus started doing a dance, and Craig's fan art of him appeared above the bus and started shaking to the beat!

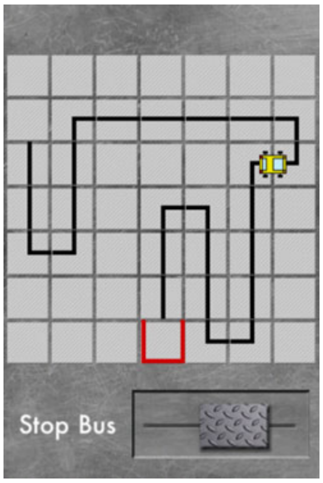


The code was very buggy and pretty damn broken. I tried to have my brother play the app, and it took him three tries to do anything at all because he would click the wrong square at the wrong time and it would basically ruin the game. And I would have to completely restart the app each time because I didn't build in a "restart" feature. But I was still pretty proud of my crappy Trainyard ripoff. Matt should pay me for it.

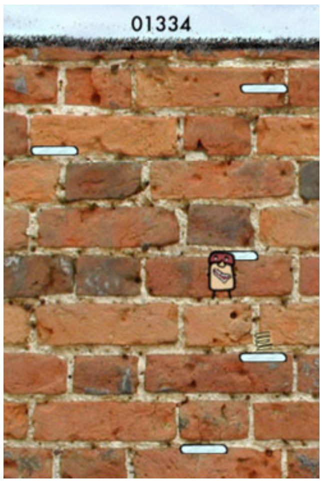
My next project was a Snake clone. I found this tutorial on how to build Snake in flash [hn.my/snake] and basically just ported it to iPhone. It gave me a lot of insight into how to make this game. Before the tutorial, I was planning on having the snake move once per frame, but was having a lot of trouble figuring out how to make his tail follow him. I was considering building up an array of all the past moves and assigning them to each of his tail pieces every tick, but that would have gotten really annoying really fast. This tutorial flipped my world upside down when I realized the Snake was not actually moving. Instead, the illusion of movement was given by simply adding and removing snake pieces from the grid on every frame. Then when you picked up an apple all it did was tell the program not to remove any pieces for five turns. I need to remake this game for sure, because the UI sucked, the controls were awful, and the code was pretty messy due to my hacked together port. But for a basic Snake game it turned out pretty well!



After Snake, I tried making Partybus 2: Partyman's Big Weekend. I got a little further than before. Ended up adding a speed slider like in Trainyard, a depot that the train could leave from, sound effects (when he crashed there was the sound of a massive car accident and he yelled "No partying today!"), and just overall cleaner code. But again, I soon ran into entangled code, and whenever I tried to fix one thing, another thing wouldn't work. My biggest problem was that when I set the speed slider all the way to its highest value, the bus would crash at a random joint in the road. This was due to the updates being called too often and the code not updating enough before the next tick. Even though I sort of knew what the problem was, it would take too much work to go back and fix it. I think I will have to make a Partybus 3: The Final Party.



My most current project is a remake of Doodle Jump. So far it's coming along quite nicely. I have platforms that keep moving higher and higher, a jumper guy, some items (well, so far only a spring), custom music and sounds, and a UI that totally rips off the original game. I'm pretty happy with this one so far. I'm going to try to keep adding a bunch of stuff to it.



Conclusion

It truly is amazing how much practice helps. Every time I write a new program, I run into a ton of unforeseen problems and bugs. While frustrating at the time, I usually plan around them the next time I program. This makes me feel like I'm in my very own while loop:

```
while (stillPrettyBadAtProgramming) {

    programmingPracticeTime++;

    Program *newProgram = [Program
programWithType:ProgramTypeGame];

    if (newProgram.isReallyGood == YES &&
programmingPracticeTime >= A_HUGE_AMOUNT)
    {
        stillPrettyBadAtProgramming = NO;
    }

}
```

I'm going to keep creating. It's been a blast so far, and I can't wait to see what the next six months entails. 🟧

Whitaker Blackall has composed music for a number of popular iOS games, including the Casey's Contraptions, Tilt to Live, Chicken Balls, Velocispider, and Super Stickman Golf. In his spare time, he is learning to program so that someday he can release a game of his own. He currently lives in Chicago with his awesome fiancée and mini Goldendoodle.

Reprinted with permission of the original author.
First appeared in *hn.my/first6*

DKIM, SPF, Mime Parsing,
Postfix, Throttling, Feedback loops…

This is what it takes to get to the inbox.
Let us deal with it.

Postmark

hacker.postmarkapp.com
Transactional email delivery for web apps.

Crash Course: Design for Startups

By PAUL STAMATIOU

I HAVE BEEN BRAINSTORMING for the past few days about how to scope this article. Unfortunately I don't think I can distill everything about design into this or any number of posts. For one, design can be very subjective (or just plain wrong at times). Second, there's a sharp distinction between graphic design and user experience that deserves its own article. Third, there's a whole world of typography, color theory, gestalt laws, Fitts' law, Hick's law, visual hierarchy, UI patterns, layout mastery, and copywriting that needs to be explored firsthand. But most importantly, I'm still learning, too. This is not a definitive guide, just a friendly pointer for startup folks getting into design.

Subtlety is Key! Except When it's Not

When I was a wee pixel pusher, I would overuse whatever graphic effect I had just learned. Text-shadow? Awesome, let's put 5px 5px 5px #444. Border-radius? Knock that up to 15px. Gradients? How about from red to black?

You can imagine how horrible everything looked. Now, my rule of thumb in most cases is applying just enough to make it perceivable, no more. This usually means no blur on text-shadow and just a 1px offset, or only dealing with gradients moving between a very narrow color range.

Then what do I mean about "except when it's not"? Take for example visual hierarchy — where to draw someone's eyes first with color, contrast, and proportion. If you are going to increase the font size of a particular element, don't increase it by 1 or 2 points. Increase it by 10. Here's a nice test: take a screenshot of your website or layout and make it 3 times smaller. Can you still see the main headline or call to action well? No? Make it bigger!

Get Inspired and Stay Organized

This is not a new concept by any means, but it bears repeating: keep the right side of your brain engaged by regularly seeking great website designs, reading about design, sketching site layouts (or anything really), and more. Whenever I see a website I like, or even just a particular element of a site, I take a screenshot and archive it. I have been doing this for at least a year with the help of LittleSnapper [hn.my/lilsnap]. Be warned, the app is a little slow.

I have amassed over 700 screenshots so far. Not a day goes by that I don't archive something nice I see online. There is an app in the Mac App Store called Galleried which is the desktop app equivalent of browsing many CSS galleries, and it helps, too.

Process

My typical site redesign process usually goes like this:

1. I get in the mindset of the target audience. Since this is your startup, this probably won't involve much research as you are likely already the domain expert and ideal customer of your product. If not, checkout the Five W's of UX: Who, What, Where, When, and Why. Keep a clear cut use case in mind throughout the redesign.

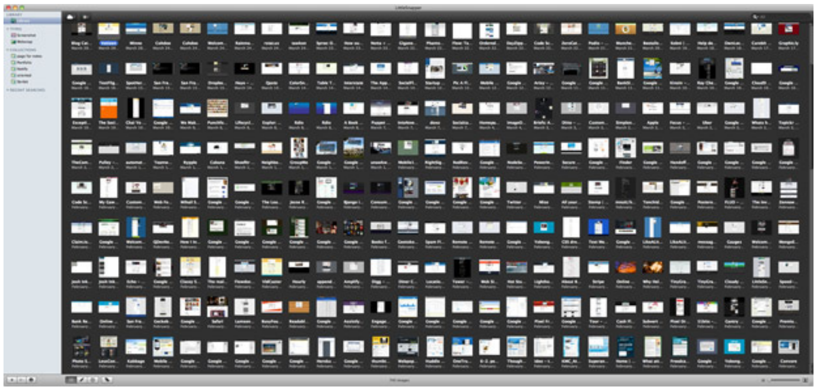
2. I spend a few hours trying to formulate my thoughts about how the first step affects layout. Should it have one big call to action for a primary use case? Are there multiple needs and products that need to be addressed? What will the hierarchy be like? What gets the most attention? I flip through the aforementioned screenshots I take every day and find 5 or 6 layouts I admire most. It might not be overall layouts, but particular elements I like.

3. The next step is either a few quick layout sketches with whatever I have on me — either my trusty Cambridge notepad and uniball Super Ink pen or UI Stencils pad — or straight to HTML/CSS. More often than not I start directly in HTML/CSS. This is a big point of contention for designers. There's a large camp of folks that start with mockups or wireframes with Photoshop, Mockingbird/Balsamiq, OmniGraffle and so on, and those that begin in markup. The points for starting with the former are that it's easier to change things on the fly, and you're more open to trying radical changes that would usually require substantial markup changes. Kyle Neath of GitHub prefers a mixture of both methods: screenshot stuff, cut it up and tweak in Photoshop, then implement.

4. I sketch 2 or 3 simple layout variants on a notepad, no more than 20 minutes, then go straight to markup. I setup Sass, import my mixins, reset, and get to work. Once a basic semblance of a website is up, I actually do a lot of design tinkering in Chrome Dev Tools. I used to staunchly prefer Firebug, but webkit and Chrome Dev Tools have come a very long way.

5. And then I take more screenshots of the Chrome Dev Tools-edited variations I like. I probably had 10 variations before I went with the one I liked for Notifo.

6. I tend to start with grayscale then tinker with color after I have the visual hierarchy down. I use xScope to help align anything and everything. It's perfect for quickly measuring space between elements (the red lines and measurements in the first image in this post is xScope).


Easy to use with their browser bookmarklet. LittleSnapper stays out of sight and saves the screenshots.

Little known fact: the Ballmer peak phenomenon doesn't only apply to coding, it works with design too. That explains the two buck chuck.

7 Color adjustments and typography tweaks usually continue through to the very last minute. I'm always experimenting to see what it would look like with other tones, hues, and font stacks.

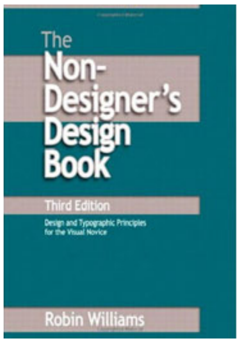
8 Most work up until now has been for the homepage. Mentally prepare a sort of style guide — modules or patterns that will be used throughout the site. Classes to use for various sidebar elements, secondary navigation, and so on. Update layout to work for content pages and style one-off pages like login, signup, and a tour or benefits page.

9 Ask around for feedback and incorporate changes.

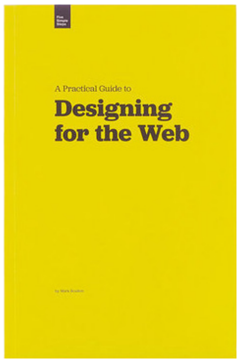
For smaller sites, like Pic A Fight, I skip all this and just wing it in HTML/CSS. On the other hand, sites like Skribit have several layouts — i.e. the homepage structure was completely different from the structure used on logged-in user pages, so there's some extra work there.

Required Reading

While I could easily recommend many great design books, such as Universal Principles of Design and Norman's classic The Design of Everyday Things, I'll start with some of what I consider to be the essentials for web and graphic design.


The Non-Designer's Design Book
Third Edition
Design and Typographic Principles for the Visual Novice
Robin Williams

Read the Non-Designer's Design Book and learn about basic color theory and C.R.A.P. — Contrast, Repetition, Alignment, and Proximity. Cool colors like blue recede, while warm colors like orange seem to move toward the viewer.



Mark Boulton's A Practical Guide to Designing for the Web is fantastic. I purchased the PDF to help support Mark, but a free online version [hn.my/dftw] is available as well. There are some sections in the book about how to get started freelancing, business plan stuff, dealing with client briefs and so on, but you can skip that. You're already well on your way with your startup.

Coders that get stumped usually do something non-code related for a while and come back to see that the code gremlins have fixed their problem, or going for a walk helped them think outside the box. Something similar applies for design. Keep your brain active by constantly seeking new sources of inspiration and creativity. I got a Wacom tablet to doodle in Illustrator and found that helps with idea generation a bunch. Yes, this is a long caption.

If you were going to only go with one design book, this would be it. Mark covers it all: type classification, typesetting, everything about color (with site examples), color as emotion, designing without color, using and not using grids, composition basics, including lead room and movement. It makes for a fairly quick read. Knock it out on a lazy analog Sunday and achieve design enlightenment.

The Elements of Typographic Style by Robert Bringhurst is widely considered to be one of the great works discussing typography, but it is mainly concerned with print. Fortunately, Richard Rutter and Steve Marshall have adapted Bringhurt's work vis-a-vis the web. The Elements of Typographic Style Applied to the Web [webtypography.net] is a great foundation in typography for the web, including various CSS snippets throughout. For example, never use `line-height` with absolute units as that can actually result in negative leading on browser font-size increases. Use a unit-less value[2] greater than 1.4 to keep leading proportional to text size.

More Homework

Get a Typekit account and rigorously browse through fonts. Take note of their organization structure. Learn about font stacks. Try out some fonts on a site of yours. Make sure you set fallback fonts. Experiment. Learn about lettering.js [letteringjs.com]. Aim for contrast while avoiding conflict. Play with font size, weight, structure, form, direction, and color. Typography is easily one of the most overlooked aspects of design for new web designers. It makes a huge difference and is worth exploring. Great designers treat text as UI has always been Cameron Moll's mantra.

Briefly read up on the Gestalt Principles [hn.my/gestalt] as they refer to user interface design: proximity, similarity, good continuation, closure, common fate, past experience, figure, and ground. And Fitts' Law [hn.my/fitts], too. Then find out what UX [hn.my/ux] really means. ■

Paul Stamatiou is the co-founder of a stealth startup currently in Y Combinator's Summer 2011 batch. You can follow him online: @Stammy

Reprinted with permission of the original author.
First appeared in *hn.my/design*

What I Learned from Fixing my Laptop's Motherboard

By DIOMIDIS SPINELLIS

A MONTH AGO, I managed to break my laptop by reversing the polarity of a universal power supply. The repair shop diagnosed the problem as a failed motherboard and asked for €659 to replace it. I found the price preposterous, and the notion of throwing away a motherboard for a single failed component ecologically unsound. Here is how I fixed the laptop on my own, and what I learned in the process.

Thankfully, I was quickly able to find a service manual. The troubleshooting guide quickly led me in the same direction as that of the repair shop: "Replace the motherboard." However, the manual also provided me the exact 28-step sequence for extracting the motherboard. It involved removing more than 40 screws, unseating about a dozen connectors, and separating a similar number of parts.

Lesson ❶ Don't attempt to disassemble a laptop without a service guide.

Lesson ❷ Keep notes on which screw belongs to which part. Place screws on paper sheets numbered by the step on which you removed them. This will help you reassembling the chaos into one working piece.

Lesson ❸ Release connectors before pulling flat cables. Usually you pull a retaining part out or up. Be careful, flat cables and their connectors are very fragile.

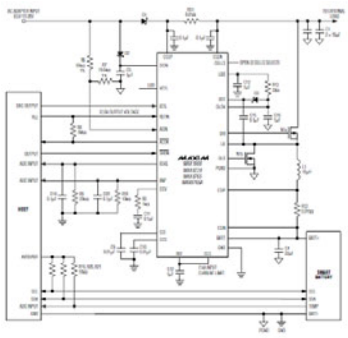
Locating the problem on the motherboard proved a lot more difficult. When I was young I remember consumer electronics, like tape recorders, coming with their circuit diagram as part of their documentation. I was fascinated by them. Later those diagrams got relegated to service manuals. The technical reference manual of the original IBM PC contained the detailed circuit diagram of every part. Sadly, all my laptop's service manual offered was a useless (for my purposes) block diagram.

I thus had to resort to other methods. I quickly learned two more lessons.

Lesson ❹ Following circuit traces on a modern multi-layer board is a futile exercise.

Lesson ❺ You can find component connections by testing logical places for connectivity with a multimeter.

This last strategy proved easy to follow once I located the key components used for the power supply subsystem: a power supply controller and a battery charger IC made by Maxim. The corresponding data sheets gave me a circuit diagram that was very close to what I saw on the motherboard.



Lesson ❻ You can reverse engineer complex systems by reading the application notes of key components used; original equipment manufacturers seem to follow closely the plans of component suppliers.

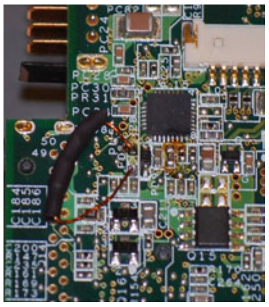
Locating the data sheet for each component on the Internet proved invaluable. I thus quickly found out that tens of what looked like 8-pin ICs were in fact MOSFETs packaged with a Schottky diode.

Lesson ❼ Data sheets are your friends.

Given that the motherboard was in theory a complete write-off, I decided to test it under power. By measuring voltage at various points, I found that a diode that was supposed to supply the battery charging controller was broken. When I short-circuited the diode and found voltage at various other motherboard locations that were previously dead, I was sure I had located the culprit.

Lesson ❽ To locate a fault, be prepared to work top-down (from subsystems to components) or bottom-up (look for a faulty component) or to start from the fault's reason (e.g. reversed polarity) or its symptom (e.g. lack of power).

Fixing the problem on a dense board of surface-mounted components wasn't trivial, however. Fortunately, finding a replacement part was easy. According to the application note, I could use a simple signal diode, so I just pulled a 1N4148 diode out of my component drawer. To fit it on the circuit board, I soldered thin insulated solid copper wire at its two ends and placed the package in a heat-shrink tube. This allowed me to place the package nearby and solder the two ends on the pads left by removing the original diode that had failed.



Lesson ❾ For the fix you'll have to improvise and make concessions. The tools of your trade are a very small soldering iron, a magnifying glass, thin wire-wrap cable, and insulation materials.

The fix proved correct, and some time later I was happily using my newly revived laptop.

Lesson ❿ Fixing a modern motherboard isn't trivial, but it isn't impossible. ■

Diomidis Spinellis is a professor in the Department of Management Science and Technology at the Athens University of Economics and Business and the author of the award-winning books Code Reading and Code Quality.

Learn to Remember Everything

The Memory Palace Technique

By RUBEN BERENGUEL

In this article I'll teach you how to have perfect recall of lists of items. Length is not much of an issue; it can be your shopping list of 10 items, or it can be a list with 50, 100, or even 1000. And in a forthcoming post I'll show you how to apply this technique to learning new languages. Sounds good, doesn't it?

The technique we'll be learning is called the memory palace and is also known as the method of loci (for the Latin word locus, meaning "place") and also the mind palace.

The Memory Palace

The memory palace technique began in the 5th century B.C., when Simonides of Ceos, poet, was attending an unfortunate banquet in Thessalia. While he was away to talk with a courier who asked for him outside, the hall's ceiling crumbled, killing everyone. There was no way to recognize the corpses...until Simonides realized that it was no problem to recall who was where without any effort.

Think about it: It is not hard to remember who sits beside the host, where your friends sit, who is beside them, and so on. This dawned upon Simonides, and he is credited as the "inventor" of the memory palace technique. Widely spread through antiquity, there were not a lot of written accounts on it: it appears in the anonymous Rhetorica ad Herrenium and Cicero's De Oratore. It is not that strange that there were no written accounts. It is like writing a book about how to put your trousers on. Everybody knows how to do it.

The memory palace is well suited to how our brains have evolved. Back in our nomadic days we needed to know how to get somewhere (the lake, the plain) and remember what was there (fresh water, hunting). By taking advantage of this fact we can build an array of impressive memorization techniques for ordered or unordered lists.

“Remembering lists may sound lame — who wants to memorize a list? But lists are just an ordered array of knowledge!”

Remembering lists may sound lame — who wants to memorize a list? But lists are just an ordered array of knowledge! What you study for a history exam is a list of ordered dates accompanied by facts and causes (sub-lists). When you learn a new recipe, it is a list. A telephone number is a list of numbers. A poem is a list of phrases.

Your First Memory Palace: Building & Filling

Let’s start by creating our first memory palace. It does not need to be a palace — in fact, it shouldn’t be. Just think of your home, and as a sample I’ll assume is really small: from the door you get to a small hall, connected to a living room which leads to a kitchen, a bathroom, and a bedroom with a balcony. This is a sample; to memorize correctly, you have to visualize your home or any other place you may know very well. You can, of course, use this mental image of an imaginary house, but memorizing may be harder, be warned.

Now consider the following shopping list: lettuce, bacon, onion rings, SD card, and oranges. We want to memorize it. I picked a short list to make the post shorter and make it fit in our small imaginary home. Try your hand with a longer list if you don’t believe we can do it with longer lists.

To remember the list, we have to place each item somewhere in our mind palace. This, of course, can mean one item per room or several items per room, each one in a special spot in the room. The simplest method is to put each item in its own room. When you are confident enough, create additional trapping space in each room. Thus, our small 5-room house could be easily a 5, 10, or 15 places memory palace.

To place an item, we have to visualize it in the room, and to make sure we remember it, it has to be an extremely odd image. It has to leave a clear impression, and to do so, it has to be surprising, bizarre, or sexual, among other options. If the image is dull, remembering it is close to impossible.

Begin with the list. When we enter the front door, we are greeted by Kermit the frog only that this special Kermit is made of lettuce, like a talking lettuce. Can you see it? Feel the freshness of Lettucit's leaves? In the living room a stampede of pigs followed by Kevin Bacon with a fork should be bizarre and clear enough! In the kitchen, Scarlett Johansson plays hoola-hoop with an onion ring. You enter the bedroom, and to your surprise, the bed is a gigantic SD card: you can hide the bed by pressing it in to be read. Finally, you open the balcony to find that the sun is now a big, luminous orange. It starts to drip juice over the desert in front of your window!

You should put all these images in a place you know like the palm of your hand: your home, the house you grew up, your office. This is very important.

You may not believe it works at all, but you will be surprised. I wrote the first part of this post in the afternoon, and now more than 3 hours later I still can see clearly all the images. Of course this is a short list... But it would not matter: you could remember a list 5 times as long as easily as with this one.

Finding an Array of Memory Palaces

To remember a lot of things, you need to have a lot of places to put all these memories. You will need to find your own array of memory palaces. The first time I considered this problem, I thought about creating imaginary palaces linked somehow by corridors. The problem? Artificial palaces get blurry very quickly, and you tend to forget them. It is far, far better to use real places, or at least places you can revisit in real life, like pictures from a book, levels in a computer game, or buildings you can visit.

Then I started to think about houses and places I could use — and I found that there are plenty. I still remember schoolmates' houses from 16 years ago, hotels I've been to, buildings I have visited. I am sure you will find a huge array of places you can use. To begin with the technique, use very known places like your house or office, and as you get more confident with the technique, start using older places.

Final Words

You have to get the knack of the method. Get some degree of experience in converting everyday objects (like lettuce) into long-lasting impressions (like Kermit the lettuce-head). This only comes with practice, like walking around your images of memory palaces. Practice, practice, practice!

By the way, can you recall the shopping list above? ■

Ruben Berenguel is a mathematician finishing his PhD about invariant manifolds in infinite dimensional dynamical systems. A lifelong language geek, he is currently trying to learn Irish and Icelandic, and setting sights on Norwegian and Swedish. He blogs in mostlymaths.net and tweets as @berenguel

The business book for entrepreneurs: over 30 interviews with startup founders.

Follow along as a Y Combinator alumni interviews other startup founders to understand how they deal with challenges such as finding cofounders, raising money, getting users, staying motivated, and hiring.

STARTUPS
OPEN SOURCED

NEW

stories to inspire & educate

Grooveshark Reddit GitHub Foursquare Airbnb Weebly Greplin AppSumo Wufoo LittleAppFactory MixPanel LikeALittle Djangy Div-vyshot Justin.TV Blippy Bump WePay DailyBooth Gobble KISSMetrics Omnisio Cloudkick Noteleaf OneLlama Octopart Crowdbooster Listia Hipmunk Indinero OrangeQC Camino Real One

Jared Tame

What people are saying

“This is great! Having interviewed over 300 founders, I’m impressed - if you want to understand how some of today’s most impressive startup founders got to where they are, this book will deliver.”
- Andrew Warner, Mixergy.com

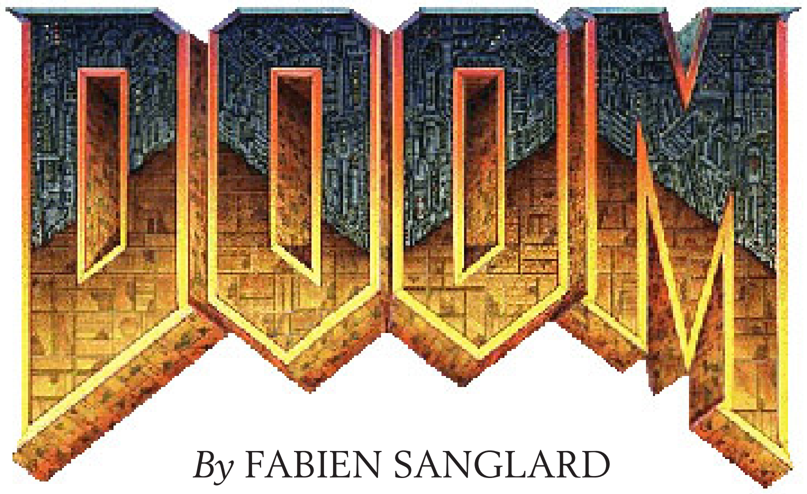
“One of the richest collections of authentic and inspiring stories.”
- Avichal Garg, BluePrint Labs cofounder

“I think this book will end up causing more students to try to become entrepreneurs.”
- Niniane Wang, Cofounder Sunfire Offices

Special discount for Hacker Monthly readers!

First 250 May edition readers receive 15% off at **StartupsOpenSourced.com**, just use coupon code “**hmonthly511**”

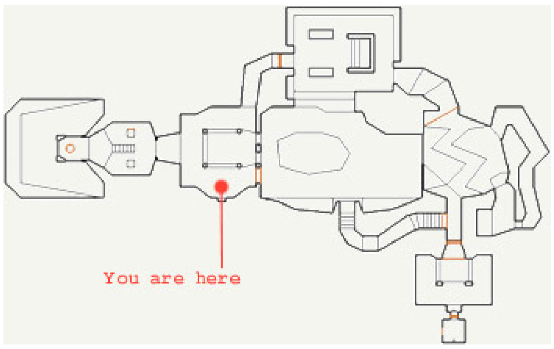
Doom Engine Code Review



By FABIEN SANGLARD

Introduction

Before studying the iPhone version, it was important for me to understand how Doom engine *was* performing rendition back in 1993. After all, the OpenGL port must reuse the same data from the WAD archive. Here are my notes about Doom 1993 renderer; maybe it will help someone to dive in.



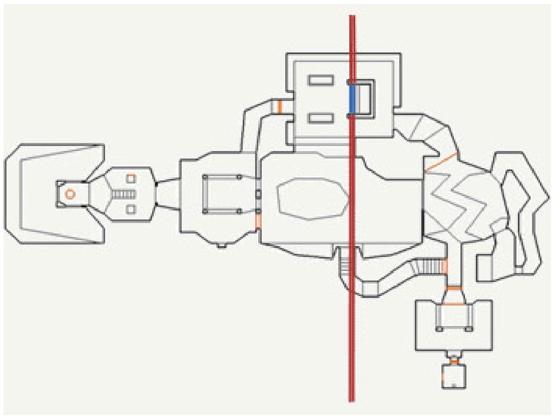
From Designer Screen to Player Screen

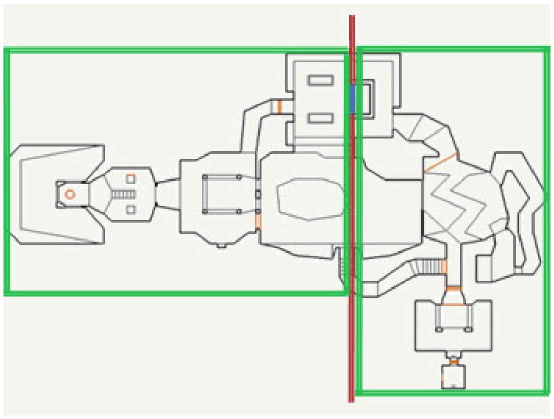
Maps were designed in 2D by a level designer using Doom editor (DoomED). LINEDEFS were describing closed sectors (SECTORS in the source code); the third dimension (height) was defined on a per sector basis. The first level of Doom E1M1 looks like this:

When the map was finished, it was sliced via Binary Space Partitioning. Recursively, a LINEDEF was chosen and its plan extended as splitting plan. LINEDEF were hence cut into segments (SEGS) until only convex SubSectors (SSECTOR in the code) remained.

Following is an example of the first map being recursively split:

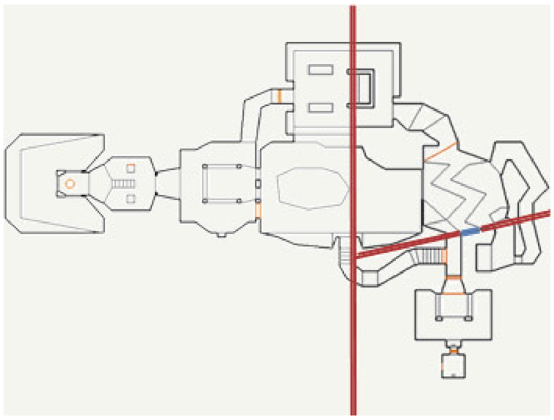
Recursion level 1

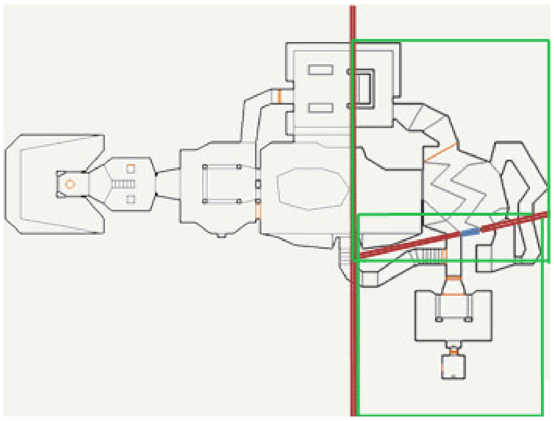




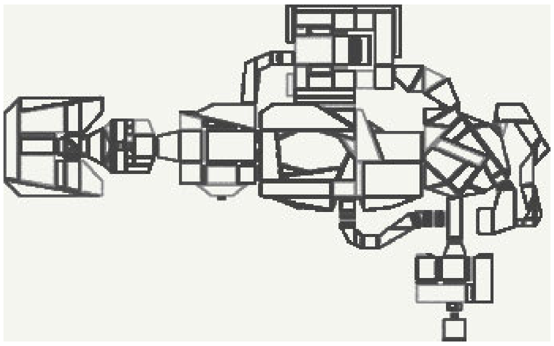
In blue, a wall is selected and extended as splitting plan (red). Splitter was selected in order to balance the BSP tree, but also to limit the number of SEGS generated. The green bounding boxes are used later to discard entire chunks of map.

Recursion level 2 (only right subspace)





In the end, SECTORS were spliced into convex sub-sectors (called SSECTORS) and LINEDEFS were sliced into segments (called SEGS):



The Big Picture of Runtime

Here is what the main rendering method (`R_RenderPlayerView`) looks like:

```
void R_RenderPlayerView (player_t* player)
{
        [..]
        R_RenderBSPNode (numnodes-1);
        R_DrawPlanes ();
        R_DrawMasked ();
}
```

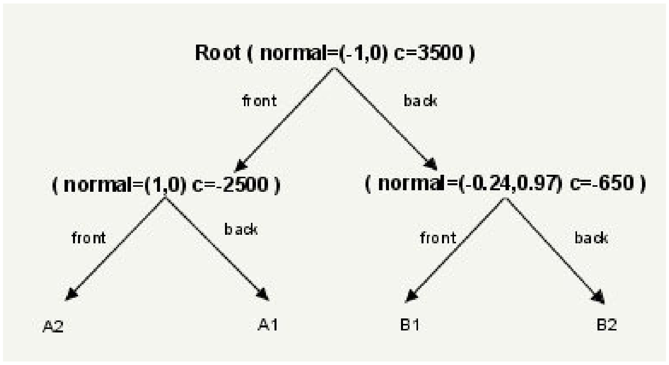
Four things happen:

- `R_RenderBSPNode`: All sub-sectors in the map are sorted using the BSP tree. Big chunks are discarded via bounding box (green in the previous drawing).
- `R_RenderBSPNode`: Visible SEGS are projected on screen via a lookup table and clipped via an occlusion array. Walls are drawn as column of pixels. The size of a column is determined by the distance from the player POV and the Y position of a column via the height relative to the played. The base and the top of the walls generate visplanes, a structure used to render the floor and ceiling (called flats).
- `R_DrawPlanes`: Visplanes are converted from column of pixels to lines of pixels and rendered to screen.
- `R_DrawMasked`: The "things" (enemies, objects, and transparent walls) are rendered.

Binary Space Partition Sorting

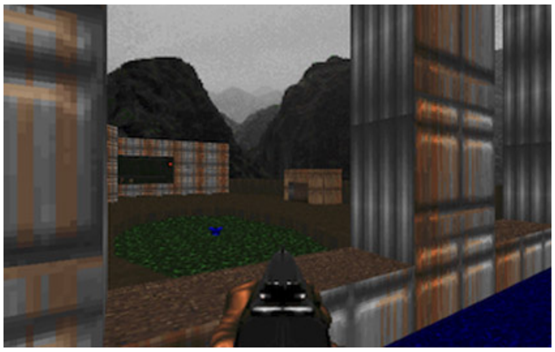
Two examples with E1M1 (Doom first map) and a BSP looking as follows:

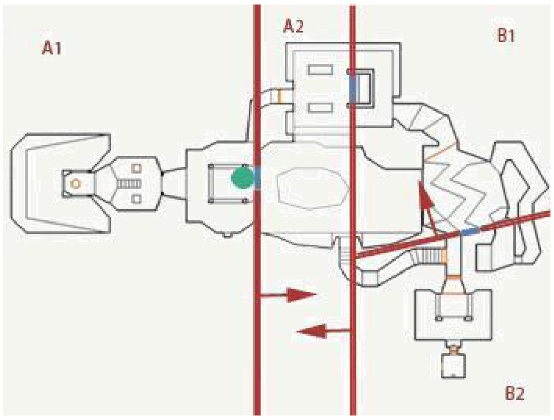
```
// Coordinate system origin in lower left
// corner

// Plane equation ax + by + c = 0 with
// unit normal vector = (a,b)

// Root plane (splitting map between zone
// A and B):

normal = (-1,0)        c = 3500

// A plane (splitting zone A between zone
// A1 and A2):

normal = (1,0)         c = -2500

// B plane (splitting zone B between zone
// B1 and B2):

normal = (-0.24,0.94)  c = -650

// Injecting any point coordinate (x,y) in
// a plane equation gives the distance
// from that plane.
```



BSP walking always starts at the root node, sorting both subspaces. Recursion follows on both node children.

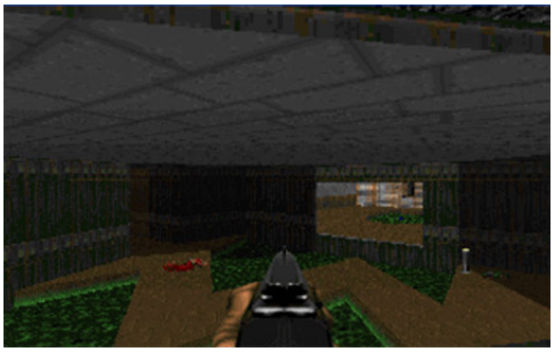
Example 1: Player (green dot) watching through the window from point p=(2300,1900):

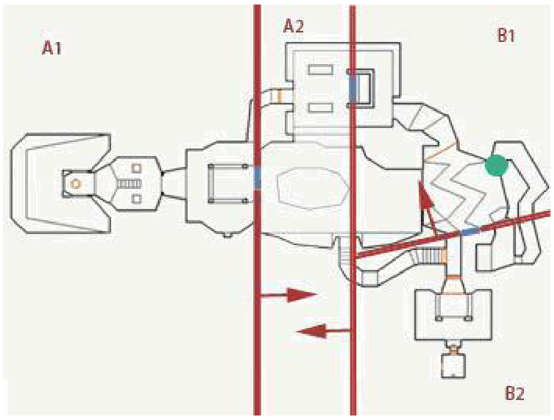




```
// Player position = ( 2300, 1900 )
// R_RenderBSPNode run against AB splitter
// (-x + 3500 = 0):
-2300 + 3500 = 1200
Result is positive: Closest subspace is in
the FRONT of the splitting plane. (A is
closer than B).

// R_RenderBSPNode now run recursively
// against the two child of the root node:
// A1/A2 splitter and B1/B2 splitter.

  // R_RenderBSPNode run against A1/A2
  // splitter (x - 2500 = 0):
  2300 - 2500 = -200
  Result is negative so the closest
  subspace is in the BACK of the splitting
  plane. (A1 is closer than A2).

  // R_RenderBSPNode run against B1/B2
  // splitter (-0.24x +0.97y - 650 = 0):
  -0.24 * 2300 + 0.97 * 1900- 650 = 641
  Result is positive so the closest
  subspace is in the FRONT of the
  splitting plane. (B1 is closer than B2).

Result: Sorted zones, from near to far:
{ A1, A2, B1, B2 }
```

Example 2: Player (green dot) watching from the secret balcony a point p=(5040, 2400):





```
// Player position = ( 5040, 2400 )
// R_RenderBSPNode run against AB splitter
// (-x + 3500 = 0):
-5040 + 3500 = -1540
Result is negative: Closest subspace is
in the BACK of the splitting plane. (B is
closer than A).
```

```
// R_RenderBSPNode now recursively run
// against the two child of the root node:
// A1/A2 splitter and B1/B2 splitter.

  // R_RenderBSPNode run against B1/B2
  // splitter (-0.24x +0.97y - 650 = 0):
  -0.24 * 5040 + 0.97 * 2400 - 650 = 468
  Result is positive so the closest
  subspace is in the FRONT of the
  splitting plane. (B1 is closer than B2).

  // R_RenderBSPNode run against A1/A2
  // splitter (x - 2500 = 0):
  5040 - 2500 = 2540
  Result is positive so the closest
  subspace is in the FRONT of the
  splitting plane. (A2 is closer than A1).

Result: Sorted zones, from near to far:
{ B1, B2, A2, A1 }
```

BSP allowed SEGS sorting from anywhere in the map at a consistent speed, regardless of the player's location. At the cost of one dot product and one addition per plane. Entire part of the map was also discarded via bounding box testing.

Note: It is not immediately apparent but BSP sorting all SEGS around the player, even the ones he/she is not looking at, frustrum culling is essential when using BSP.

Walls

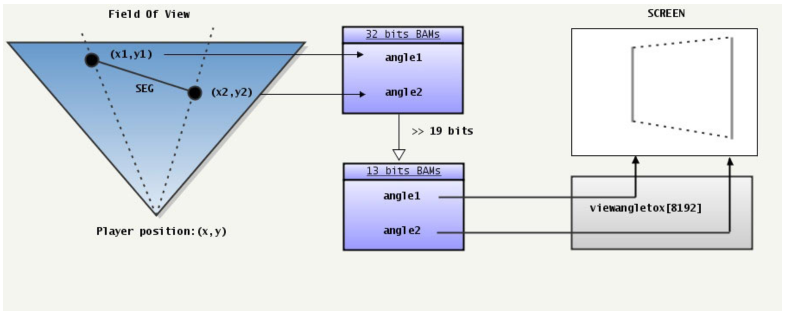
With the BSP sorting walls (SEGS) near to far, the closest 256 walls were rendered. Every SEGS's two vertices were converted to two angles (relative to the player's position).

Note: In 1993, only the very high-end 486DX machines had a FPU (floating point unit), hence Doom engine was doing all angles calculation via Binary Angular Measurement (BAMs), relying on int only, float is rarely used. For the same reason, sub integer precision is achieved via `fixed_t` a 16.16 binary fixed point.

Once converted to angle screen space X coordinate are retrieved via a lookup table (`viewangletox`). Because BAMs were int, angles were first scaled down from 32 bits to 13 bits via a 19 bits right shift in order to fit the 8k lookup table.

The wall is then clipped against an occlusion array (`solidsegs`, some articles about Doom engine mention a linked list but it does not look like it). After clipping, space remaining was interpolated and drawn as column of pixels: the height and Y coordinate of the column of pixel were based respectively on the SEGS's sector height and its distance from player POV.

Note about surface culling: Backface culling was done via `angle2-angle1 > 180` . Only walls within the Field of View were actually rendered.



Note: Not all walls were made of an unique texture; walls could have a lower texture, a upper texture, and a middle texture (that could be transparent or semi-transparent).

Trivia: Because walls were rendered as columns, wall textures were stored in memory rotated 90 degrees to the left. This was done to reduce the amount of computation required for texture coordinates:

```
int WIDTH;
int HEIGHT;

char texture[WIDTH*HEIGHT];

char* firstPixelInColumn = texture;
char* lastPixelInColumn ;

// If the texture is stored vertically in
// memory, the last element in a column is:
lastPixelInColumn = firstPixelInColumn +
textureWidth * (HEIGHT-1);

// If the texture is stored horizontally in
// memory, the last element in a column is:
lastPixelInColumn = firstPixelInColumn +
HEIGHT-1;
```

Flats (Ceiling and Floor) or the Infamous Visplanes

While drawing column of walls, top and bottom screen space coordinates were used to generate "visplanes," an area in screen space (not necessarily continuous horizontally). Here is a visplane_t as declared in Doom engine.

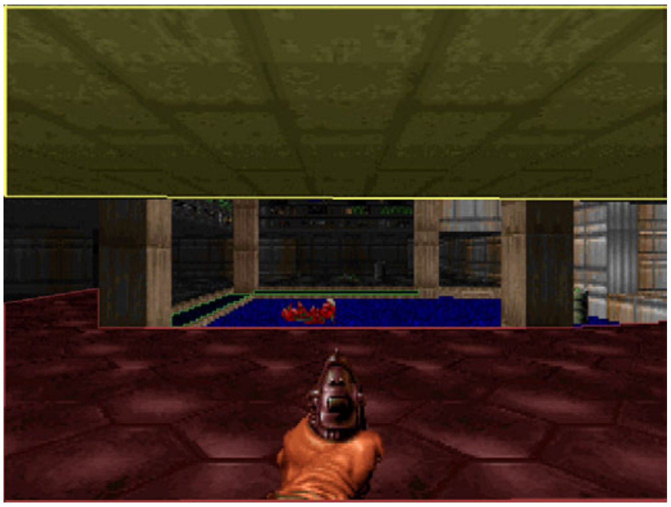
```
// Now what is a visplane, anyway?
typedef struct
{
        fixed_t         height;
        int             picnum;
        int             lightlevel;
        int             minx;
        int             maxx;
        byte            top[SCREENWIDTH]
        byte            bottom[SCREENWIDTH];
} visplane_t;
```

The first part of the structure holds information about the "material," (`height picnum lightlevel`). The last 4 members define the screenspace zone covered.

If 2 subSectors shared the same material (height, texture, and illumination level), Doom engine tried to merge them together, but because of the `visplante_t` structure limitation it was not always possible.

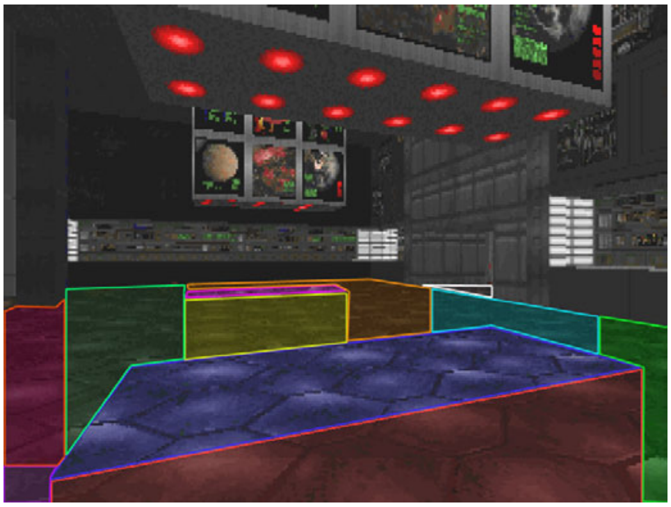
For the entire width of the screen, a visplane can store the location of a column of pixels (because visplanes are deduced from the walls projection on screen, they are created as column of pixels).

Here are the starting screen's 3 main visplanes:



The green one is particularly interesting as it illustrates `visplane_t`'s ability to store discontinued areas (but only horizontally). As long as the column is continuous, visplane can store it. This limitation shows in the engine: some subsectors can be merged and rendered via only 1 visplane, but if something comes between vertically they cannot be merged.

Here is a screenshot illustrating visplane fragmentation.



Trivia: Visplanes hardcoded limit (`MAXVISPLANES` 128) was a plague for modders as the game would crash and go back to DOS. Two issues could arise:

- "`R_FindPlane: no more visplanes`": The total number of different visplanes materials (height, texture, and illumination level) is over 128.
- `R_DrawPlanes: visplane overflow (%i)`: Visplanes fragmentation is important and number of visplanes is over 128.

Why limit it to 128? Two stages in the renderer pipeline were requested to search in the list of visplanes (via `R_FindPlane`). This was done via linear search, and it was probably too expensive beyond 128. Lee Killough later lifted this limitation, replacing linear search with a chained hash table implementation.

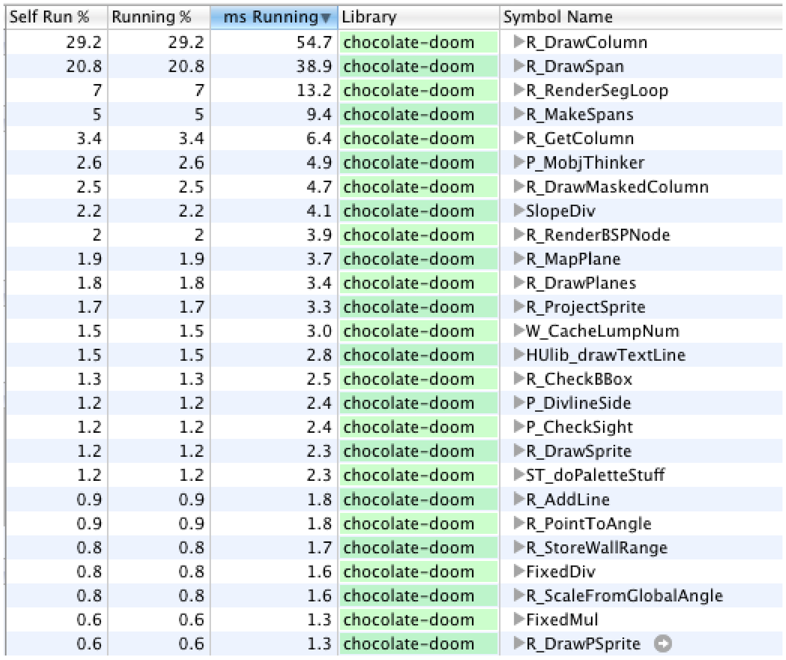
Things and Transparent Walls

When all solid/"middle texture transparent" walls and ceiling/floors surfaces were rendered, there only remained the "things," regrouping enemies, barrel, ammos, and semi-transparent walls. Those are rendered far to near but are not projected into screen space using the wall's lookup table. It's all done with 16.16 binary fixed point calculations.

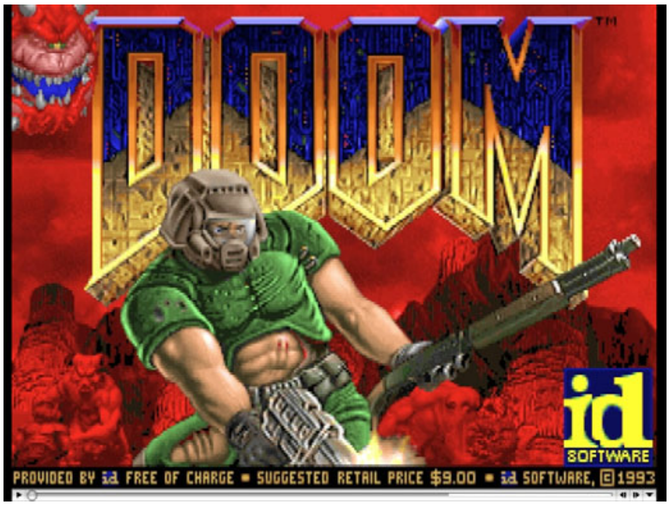
Profiling

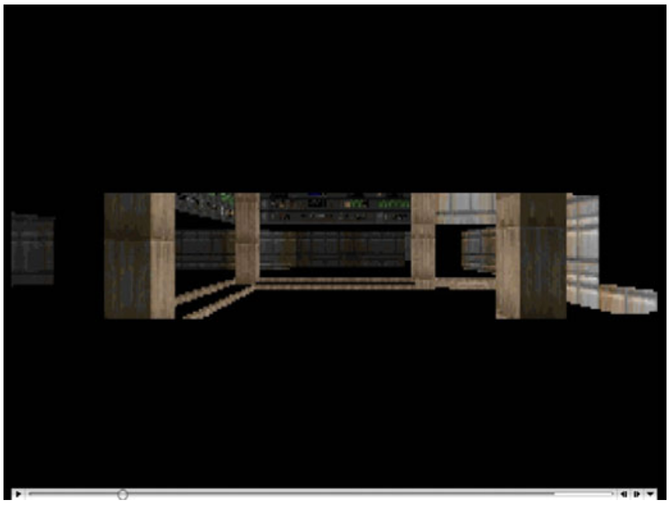
Loading "Chocolate Doom" in Mac OS X's Instrument allows you to do some profiling:

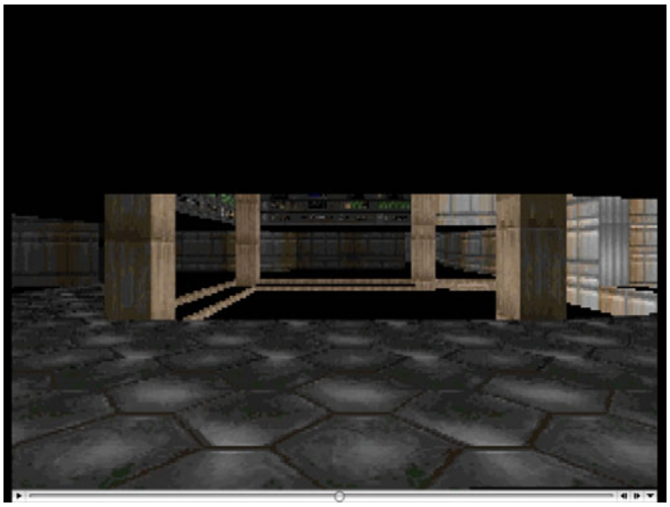
It seems the port is pretty fidele to Vanilla Doom: most time is spent drawing walls (`R_DrawColumn`), ceiling/floor(`R_DrawSpan`), and things (`R_DrawMaskedColumn`). Besides drawing, I noticed the high cost of wall interpolation (`R_RenderSegLoop`) and visplane conversion from columns to lines of pixels (`R_MakeSpans`). Finally come monsters IA (`R_MobjThinker`) and BSP traversal (`R_RenderBSPNode`).

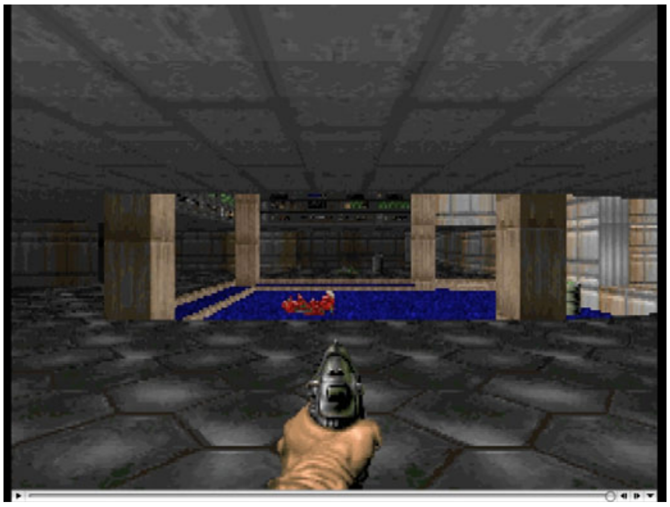


Self Run %	Running %	ms Running ▾	Library	Symbol Name
29.2	29.2	54.7	chocolate-doom	▶R_DrawColumn
20.8	20.8	38.9	chocolate-doom	▶R_DrawSpan
7	7	13.2	chocolate-doom	▶R_RenderSegLoop
5	5	9.4	chocolate-doom	▶R_MakeSpans
3.4	3.4	6.4	chocolate-doom	▶R_GetColumn
2.6	2.6	4.9	chocolate-doom	▶P_MobjThinker
2.5	2.5	4.7	chocolate-doom	▶R_DrawMaskedColumn
2.2	2.2	4.1	chocolate-doom	▶SlopeDiv
2	2	3.9	chocolate-doom	▶R_RenderBSPNode
1.9	1.9	3.7	chocolate-doom	▶R_MapPlane
1.8	1.8	3.4	chocolate-doom	▶R_DrawPlanes
1.7	1.7	3.3	chocolate-doom	▶R_ProjectSprite
1.5	1.5	3.0	chocolate-doom	▶W_CacheLumpNum
1.5	1.5	2.8	chocolate-doom	▶HUlib_drawTextLine
1.3	1.3	2.5	chocolate-doom	▶R_CheckBBox
1.2	1.2	2.4	chocolate-doom	▶P_DivlineSide
1.2	1.2	2.4	chocolate-doom	▶P_CheckSight
1.2	1.2	2.3	chocolate-doom	▶R_DrawSprite
1.2	1.2	2.3	chocolate-doom	▶ST_doPaletteStuff
0.9	0.9	1.8	chocolate-doom	▶R_AddLine
0.9	0.9	1.8	chocolate-doom	▶R_PointToAngle
0.8	0.8	1.7	chocolate-doom	▶R_StoreWallRange
0.8	0.8	1.6	chocolate-doom	▶FixedDiv
0.8	0.8	1.6	chocolate-doom	▶R_ScaleFromGlobalAngle
0.6	0.6	1.3	chocolate-doom	▶FixedMul
0.6	0.6	1.3	chocolate-doom	▶R_DrawPSprite

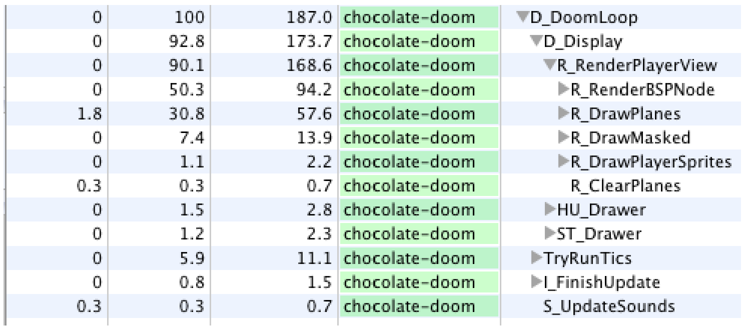

DOOM
id SOFTWARE
PROVIDED BY id FREE OF CHARGE • SUGGESTED RETAIL PRICE $9.00 • id SOFTWARE, ©1993







With an inverted call tree, we can see that most of the work is indeed done in the BSP traversal, wall rendition, and visplanes generation: R_RenderBSPNode (second column is for percentage of time).



0	100	187.0	chocolate-doom	▼D_DoomLoop
0	92.8	173.7	chocolate-doom	▼D_Display
0	90.1	168.6	chocolate-doom	▼R_RenderPlayerView
0	50.3	94.2	chocolate-doom	▶R_RenderBSPNode
1.8	30.8	57.6	chocolate-doom	▶R_DrawPlanes
0	7.4	13.9	chocolate-doom	▶R_DrawMasked
0	1.1	2.2	chocolate-doom	▶R_DrawPlayerSprites
0.3	0.3	0.7	chocolate-doom	R_ClearPlanes
0	1.5	2.8	chocolate-doom	▶HU_Drawer
0	1.2	2.3	chocolate-doom	▶ST_Drawer
0	5.9	11.1	chocolate-doom	▶TryRunTics
0	0.8	1.5	chocolate-doom	▶I_FinishUpdate
0.3	0.3	0.7	chocolate-doom	S_UpdateSounds

All Together

Finally, a video (screenshot above) of the legendary first screen generation where you can see in order:

- Walls, near to far, as column of pixels.
- Flats, near to far, as lines of pixels.
- Things, far to near. ■

Fabien Sanglard is a game developer and a technical writer specializing in 3D engines. Since 2007, his website [fabiensanglard.net] has published numerous computer graphic tutorial and engine internals reviews. Born and raised in France, he now resides in Toronto and holds a Master of Science in Computer Sciences.

Why Data Structures Matter

By JOEL NEELY

OUR EXPERIENCE ON Day 0 of JPR11 [hn.my/jpr11] yielded a nice example of the need to choose an appropriate implementation of an abstract concept. We experimented with Michael Barker's Scala implementation [hn.my/jpr11dojo] of Guy Steele's parallelizable word-splitting algorithm [hn.my/wordsplit] (slides 51-67). Here's the core of the issue.

Given a type-compatible associative operator and sequence of values, we can fold the operator over the sequence to obtain a single accumulated value. For example, because addition of integers is associative, addition can be folded over the sequence:

1, 2, 3, 4, 5, 6, 7, 8

from the left:

((((((1 + 2) + 3) + 4) + 5) + 6) + 7) + 8

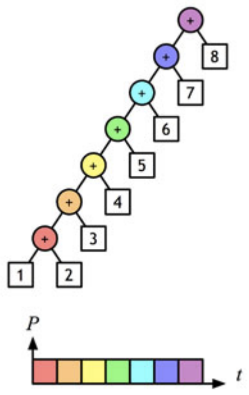
or the right:

1 + (2 + (3 + (4 + (5 + (6 + (7 + 8))))))

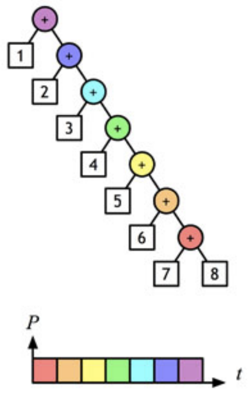
or from the middle outward, by recursive/parallel splitting:

((1 + 2) + (3 + 4)) + ((5 + 6) + (7 + 8))

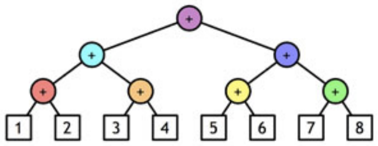
A 2-D view shows even more clearly the opportunity to evaluate sub-expressions in parallel. Assuming that addition is a constant-time operation, the left fold:

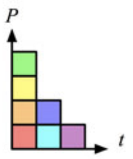


and the right fold:



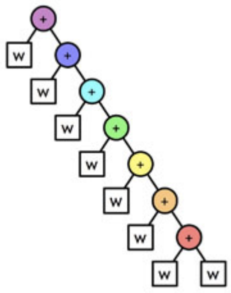
require linear time, but the balanced tree:

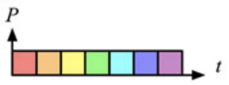




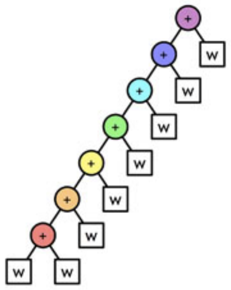
can be done in logarithmic time.

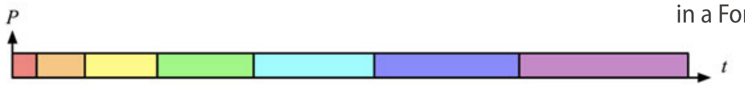
But the associative operation for the word-splitting task involves accumulating lists of words. With a naive implementation of linked lists, appending is not a constant-time operation; it is linear on the length of the left operand. So for this operation the right fold is linear on the size of the task:



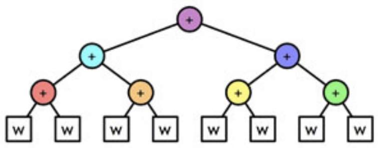


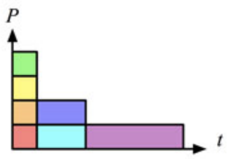
the left fold is quadratic:



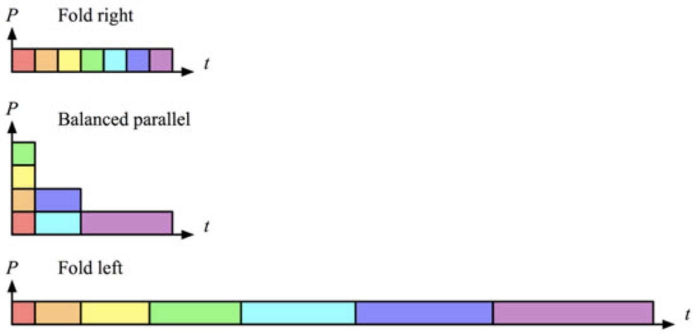


and the recursive/parallel version is linear:





Comparing just the "parallel-activity-versus-time" parts of those diagrams makes it clear that right fold is as fast as the parallel version, and also does less total work:



Of course, there are other ways to implement the sequence-of-words concept, and that is the whole point. This little example provides a nice illustration of how parallel execution of the wrong implementation is not a win. ■

Joel Neely has been programming since 1968, and has been involved in computing in higher education and industry ever since. As a survivor of Modular Programming, Structured Programming, Fifth-Generation Programming, and Object-Oriented Programming, he is watching the gradual mainstreaming of Functional Programming and multi-language development with a sense of deja vu. He currently works in development in a Fortune 100 company.

The Worst Algorithm in the World?

By ROBIN HOUSTON

YOU KNOW THE Fibonacci numbers: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, … Each number is the sum of the previous two. Let's say the zeroth Fibonacci number is zero, so:

$$\begin{aligned}\mathrm{fib}(0) &= 0\\ \mathrm{fib}(1) &= 1\\ \mathrm{fib}(n+2) &= \mathrm{fib}(n) + \mathrm{fib}(n+1)\end{aligned}$$

And let's say you want to write some code to compute this function. How would you do it? Perhaps something like this? (Python code)

```
def fib_rec(n):
  assert n >= 0
  if n < 2: return n
  return fib_rec(n-2) + fib_rec(n-1)
```

This is a pretty popular algorithm, which can be found in dozens of places online. It is also a strong candidate for the title of Worst Algorithm in the World. It's not just bad in the way that Bubble sort [hn.my/bubble] is a bad sorting algorithm; it's bad in the way that Bogosort [hn.my/bogo] is a bad sorting algorithm.

Why so bad? Well, mainly it is quite astonishingly slow. Computing `fib_rec(40)` takes about a minute and a half on my computer. To see why it's so slow, let's calculate how many calls are made to the `fib_rec` routine. If we write $c(n)$ for the number of calls made when calculating `fib_rec(n)`, then:

$$\begin{aligned}c(0) &= 1\\ c(1) &= 1\\ c(n+2) &= 1 + c(n) + c(n+1)\end{aligned}$$

So $c(n)$ are the Leonardo numbers [hn.my/leornado], $c(n) = 2\,\mathrm{fib}(n+1) - 1$ In other words, computing `fib` using `fib_rec` takes time `O(fib(n))`.

So computing `fib_rec(40)` involves `c(40)` = 331,160,281 calls to the `fib_rec` routine, which is pretty crazy considering it's only called with 40 different arguments. An obvious idea for improvement is to check whether it's being called with an argument that we've seen before, and just return the result we got last time.

```
cached_results = {}
def fib_improved(n):
  assert n >= 0
  if n < 2: return n
  if n not in cached_results:
    cached_results[n] = fib_improved(n-2) +
    fib_improved(n-1)
  return cached_results[n]
```

That's a huge improvement, and we can compute `fib_improved(40)` in a fraction of a millisecond, which is much better than a minute and a half. What about larger values?

```
>>> fib_improved(1000)
434665576869374564356885276750406258025646
605173717804024817290895365554179490518904
038798400792551692959225930803226347752096
896232398733224711616429964409065331879382
98969649928516003704476137795166849228875L
```

That looks good, and it's still apparently instant. How about 10,000?

```
>>> fib_improved(10000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in fib_improved
  File "<stdin>", line 6, in fib_improved
  [...]
  File "<stdin>", line 6, in fib_improved
RuntimeError: maximum recursion depth
exceeded
```

Oh dear! We've blown the stack. You could blame Python here for having a hard (if configurable) limit on stack size, but that's not the real point. The problem here is that this algorithm creates n stack frames when you call `fib_improved(n)`, so it uses at least `O(n)` space.

It's easy enough to write a version that only uses constant space — well, not really: but it only uses twice as much space as we need for the end result, so it's within a small constant factor of optimal — as long as we're willing to forgo recursion:

```
def fib_iter(n):
  assert n >= 0
  i, fib_i, fib_i_minus_one = 0, 0, 1
  while i < n:
    i, fib_i, fib_i_minus_one = i + 1,
fib_i_minus_one+fib_i, fib_i
  return fib_i
```

This is much better. We can compute `fib_iter(100,000)` in less than a second (on my computer, again), and fib_iter(1,000,000) in about 80 seconds. Asymptotically, this algorithm takes $O(n^2)$ time to compute `fib(n)`.

(Maybe you think it should be `O(n)` time, because the loop runs for n iterations. But the numbers we're adding are getting bigger exponentially, and you can't add two arbitrarily-large numbers in constant time. Sometimes computer scientists use theoretical models that assume you can, which makes me irrationally angry: what use is a model whose assumptions hide a physical impossibility? Of course two machine words can be added in constant time, but we're talking about asymptotic behavior, and arbitrarily large numbers don't fit in a single machine word. When I rule the world, this will be punishable.)

We can do better than this. Since the Fibonacci numbers are defined by a linear recurrence, we can express the recurrence as a matrix, and it's easy to verify that

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n = \begin{pmatrix} \mathrm{fib}(n-1) & \mathrm{fib}(n) \\ \mathrm{fib}(n) & \mathrm{fib}(n+1) \end{pmatrix}$$

Since we can get from $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n$ to $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{2n}$ by squaring, this suggests we can compute `fib(n)` using just $\log_2(n)$ iterations:

```
def bits(n):
  """Represent an integer as an array of
  binary digits.
  """
  bits = []
  while n > 0:
    n, bit = divmod(n, 2)
    bits.append(bit)
  bits.reverse()
  return bits

def fib_mat(n):
  assert n >= 0
  a, b, c = 1, 0, 1
  for bit in bits(n):
    a, b, c = a*a + b*b, a*b + b*c, b*b +
c*c
    if bit: a, b, c = b, c, b+c
  return b
```

This is certainly much faster in practice. We can compute `fib_mat(1,000,000)` in less than a second and a half. The asymptotic complexity depends on the multiplication algorithm used. If it's conventional long multiplication then multiplying two `k`-bit numbers takes time $O(k^2)$, in which case this algorithm is actually still quadratic! I believe Python uses the Karatsuba algorithm [hn.my/karatsuba], which makes it about $O(n^{1.6})$ in Python.

While we're writing code, let's improve the constant factor. Each step of `fib_mat` uses six multiplications, but we can halve that just by observing that `c` can always be computed as `a+b` and rearranging things a little:

```
def fib_fast(n):
  assert n >= 0
  a, b, c = 1, 0, 1
  for bit in bits(n):
    if bit: a, b = (a+c)*b, b*b + c*c
    else:   a, b = a*a + b*b, (a+c)*b
    c = a + b
  return b
```

And this does indeed run about twice as fast. Further improvement is possible, but I think the point has been made, so let's leave it there. If you want to see a super-efficient version, have a look at the algorithm in GMP.

Some other good articles on the subject: David Eppstein's [hn.my/eppstein] lecture notes cover similar ground to this; Peteris Krumins measures the running time of `fib_iter` [hn.my/catonmat], and explains why it's quadratic rather than linear. ■

Robin enjoys clever algorithms, good coffee, category-theoretic logic, and laughing. He lives in London, where he develops websites for mySociety.

HARVEST

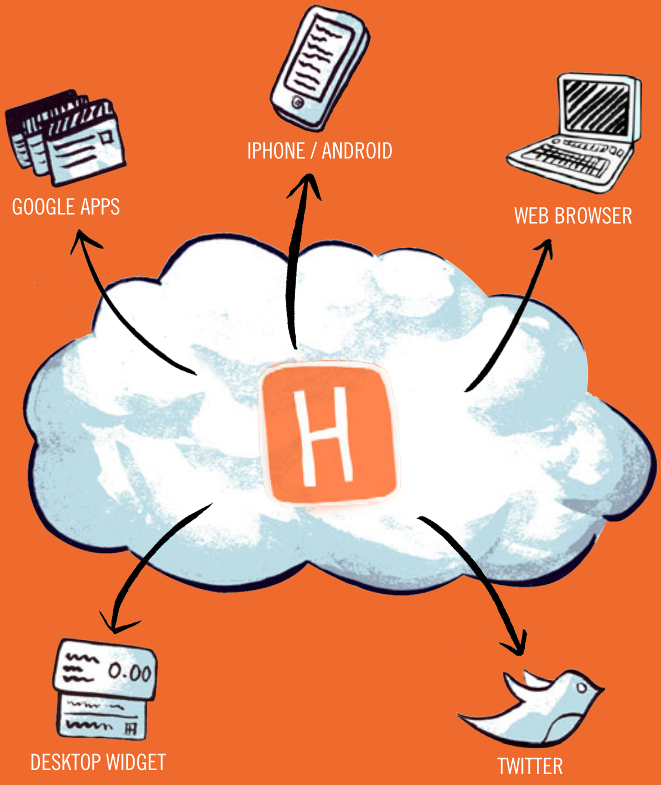
TIMESHEETS INVOICES REPORTS

Track time anywhere, and invoice your clients with ease.

Harvest is available wherever your work takes you. Whether you are working from home, on-site, or through a flight. Harvest keeps a handle on your billable time so you can invoice accurately. Visit us and learn more about how Harvest can help you work better today.

Why Harvest?

- Convenient and accessible, anywhere you go.
- Get paid twice as fast when you send web invoices.
- Trusted by small businesses in over 100 countries.
- Ability to tailor to your needs with full API.
- Fast and friendly customer support.



Learn more at **www.getHarvest.com/hackers**

Dream. Design. Print.

MagCloud, the revolutionary new self-publishing web service by HP, is changing the way ideas, stories, and images find their way into peoples' hands in a printed magazine format.

HP MagCloud capitalizes on the digital revolution, creating a web-based marketplace where traditional media companies, upstart magazine publishers, students, photographers, designers, and businesses can affordably turn their targeted content into print and digital magazine formats.

Simply upload a PDF of your content, set your selling price, and HP MagCloud takes care of the rest—processing payments, printing magazines on demand, and shipping orders to locations around the world. All magazine formatted publications are printed to order using HP Indigo technology, so they not only look fantastic but there's no waste or overruns, reducing the impact on the environment.

Become part of the future of magazine publishing today at www.magcloud.com.

25% Off the First Issue You Publish

Enter promo code **HACKER** when you set your magazine price during the publishing process.

Please contact promo@magcloud.com with any questions.

MAGCLOUD