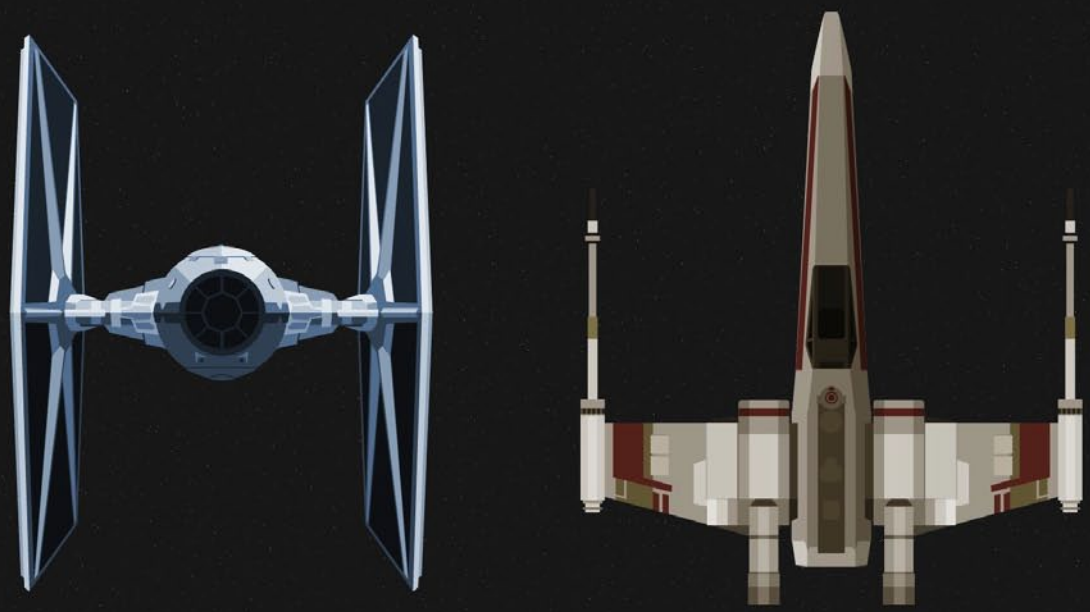


What I Learned Coding X-Wing vs. TIE Fighter

Peter Lincroft

HACKERMONTHLY

Issue 48 May 2014

Curator
Lim Cheng Soon

Contributors
Fejes Jozsef
Peter Lincroft
Steli Efti
Bobby Grace
Michael Bromley
Alvaro Castro-Castilla
James Greig

Proofreaders
Emily Griffin
Sigmarie Soto

Illustrator
Thong Le

Ebook Conversion
Ashish Kumar Jha

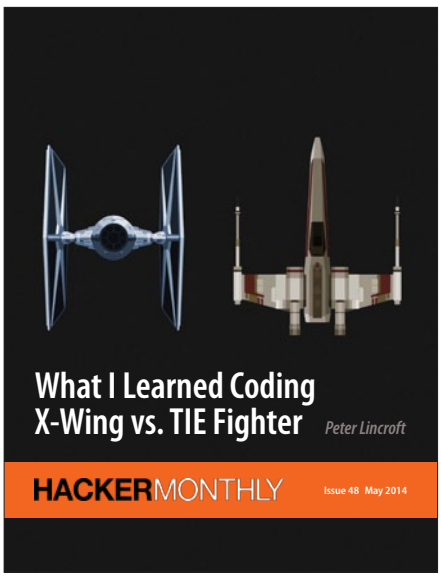
Printer
MagCloud

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

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

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



Cover Illustration: Thong Le [weaponix.net]

Contents

Illustration by Thong Le [weaponix.net]

For links to Hacker News dicussions, visit *hackermonthly.com/issue-48*

ALL RGB COLORS IN ONE IMAGE

BY FEJES JOZSEF

I RECENTLY STARTED VISITING the programming puzzles SE site. [codegolf.stackexchange.com] To a geek like me, it's a little paradise: many interesting challenges, many interesting solutions, many like-minded people. Two days ago, there was one particular challenge: make a program to create an image that contains all RGB colors exactly once (and of course the best looking one wins). A very long time ago I made a small screen saver in assembly which grew a colorful coral (I may post that one day, too). I thought something similar would work here and maybe I'll even get some votes. You can see the very first image right here. The results completely blew my mind, they were absolutely stunning, and of course it was a big success. Then I thought, let's make a huge image, maybe even a YouTube video from this. But it wasn't easy because it's a brutally exponential problem. Two days of non-stop coding and minimum sleeping later, here it is!

Of the four most widespread image formats, the images have to be in PNG format, because it supports lossless compression and all RGB colors. GIF doesn't work because it only supports at most 256 colors. JPEG doesn't work because it uses a lossy compression, so some colors are slightly altered. We need to have 100% accuracy to represent all these different colors. BMP's would be fine but they don't do any (decent) compression. So all the images you see below are PNG's, the original, raw files, which were produced by my program. Feel free to count the colors in them.

The first images: 15 bits

Let's start with some of the first images I made. These contain 15-bit RGB colors, with a resolution of 256×128 (about 32 thousand pixels). (Note: when the number of pixels in an image is a power of 2, then the dimensions of the image must also be powers of 2, but my size choices are not the only ones possible.) Little tweaks in the algorithm lead to somewhat or very different images. There are endless possibilities which I'm sure some other people will find themselves if they try it. These are just a very few samples.

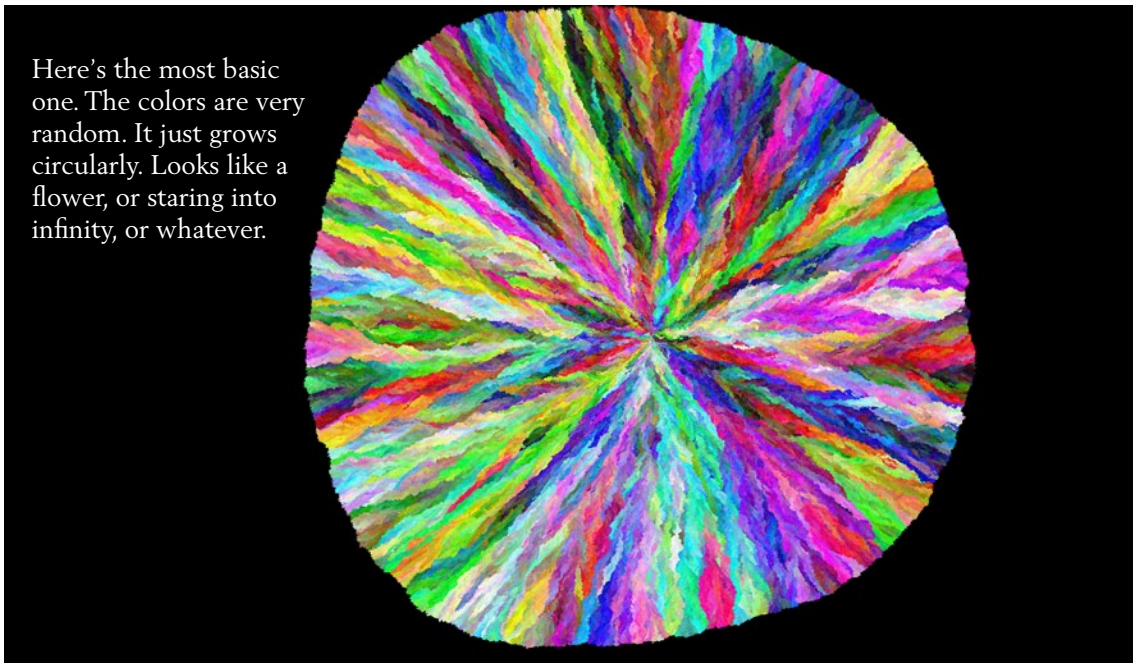
A bit bigger: 18 bits

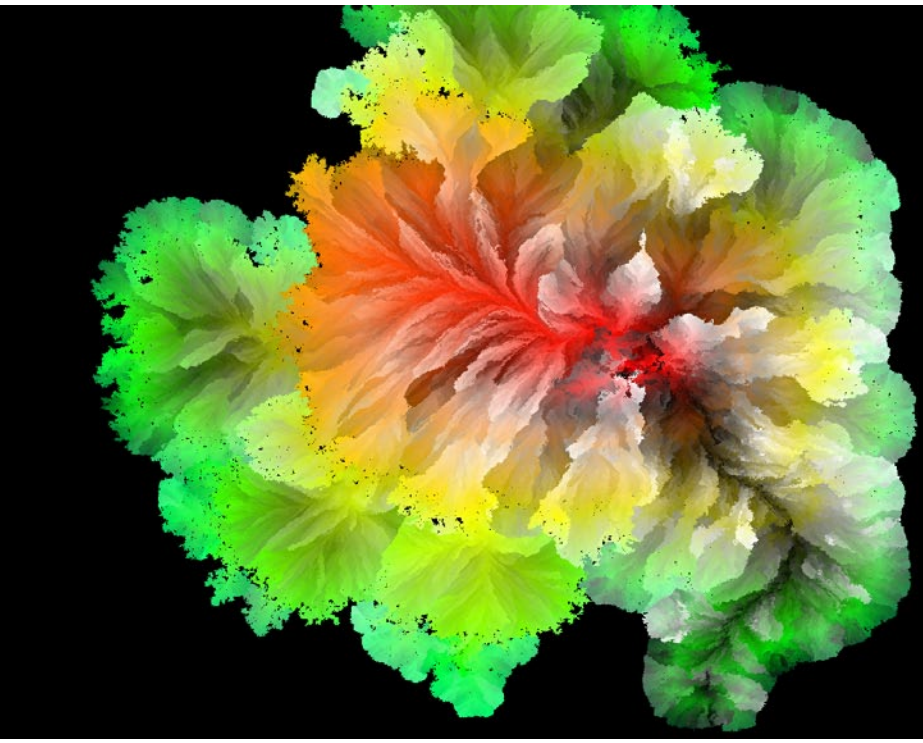
All 18 bit RGB colors fit on a 512×512 image (about 260 thousand pixels). It took tens of minutes to render them with the very first version of my program, now it only takes a few seconds. The last three images are used in the YouTube video.

Let's go to YouTube: 21 bits

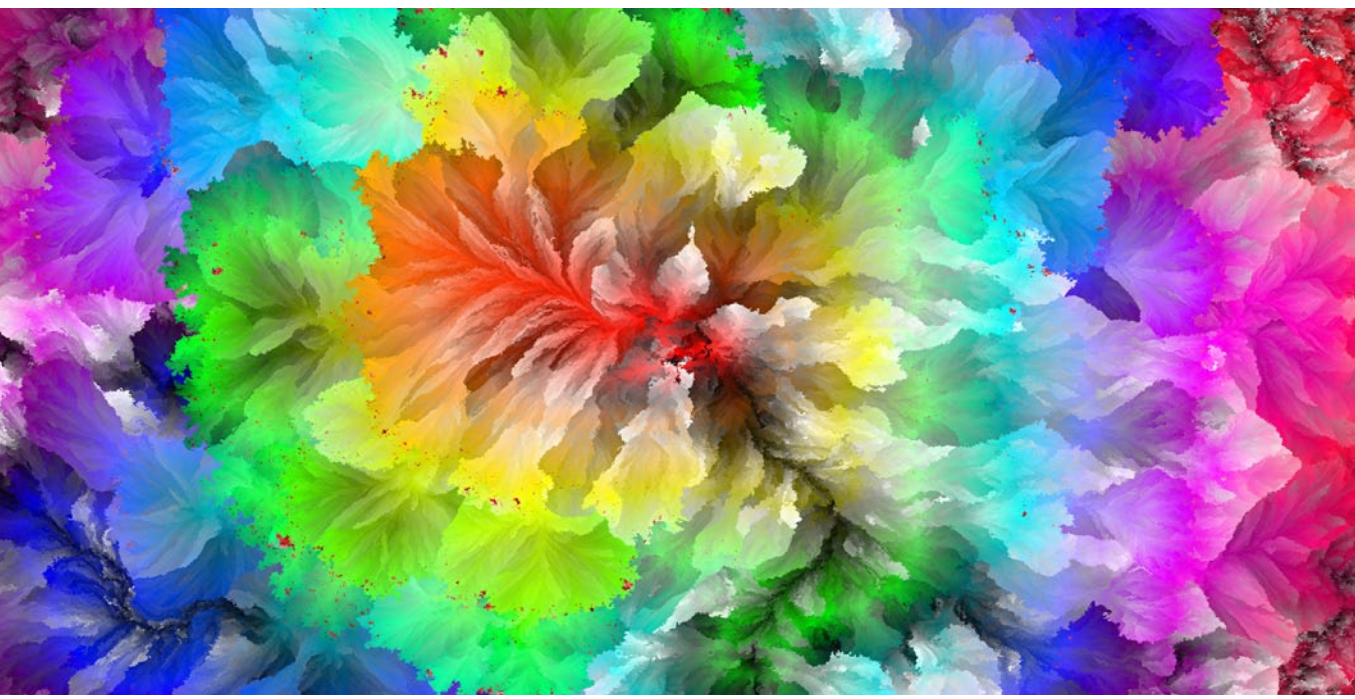
Next up: 21 bits, that's 2048×1024 (about 2 million pixels). I seriously had to work on those. Even with all the optimizations I could think of (yet!), some of these took up to 8 hours to render. Let me show you four different ones, each in an intermediate state and in the final state.

Here's the most basic one. The colors are very random. It just grows circularly. Looks like a flower, or staring into infinity, or whatever.

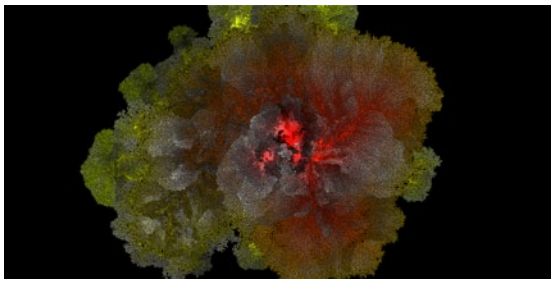




This is actually my favorite. It differs from the previous one because the colors are not entirely random, they are sorted by hue. The phrases that come to my mind are rainbow smoke and spilled ink. This is the best one to see in motion.



When you look at the end result, this one looks very much like the first one, only a little bit blurred. What's very different is how it grows. It uses a different algorithm that makes it look like a coral. The best is when it is 90-95% finished, as shown here.



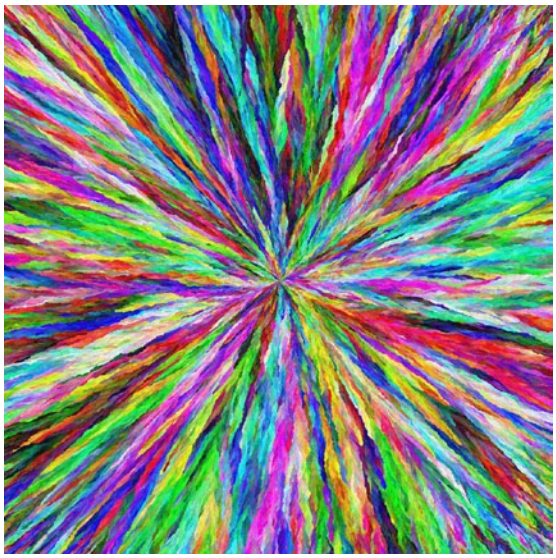
And finally, a blend between the second and third: ordered by hue, growing like a coral. When you look at the intermediate one, be sure that it's shown 1:1 and not resized because that looks bad. The end result is just weird.

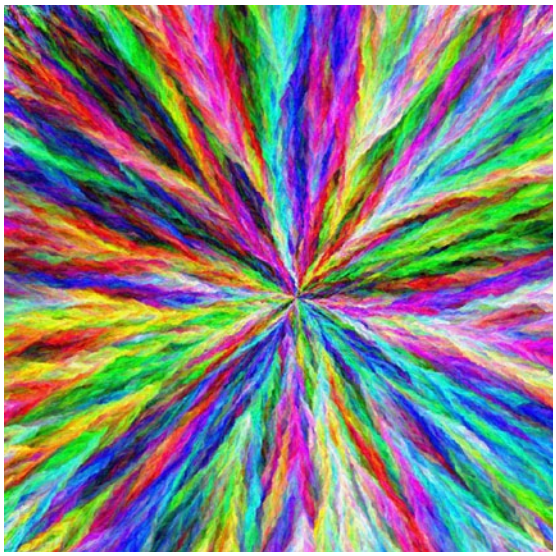
My best idea about this topic was to create a YouTube video of how each of the above four pictures are created inside the program. I never created many interesting videos before, but I knew that this is going to be a hit. Sadly the images had to be cropped (2048->1920) and I added little black bars on the top and bottom (1024->1080), and of course the video compression alters the colors slightly (and in the case of YouTube, it also introduced some really ugly artifacts). So this is not technically all-RGB imagery anymore, but it's not the point. The point is that this video looks spectacular! I also wrote a lot of source code to render the video frames, then I used FFMpeg to put it all together. Finally I chose a music from YouTube's free audio library. Enough talking, watch the video now, and be sure to watch it in HD! [hn.my/rgbvideo]

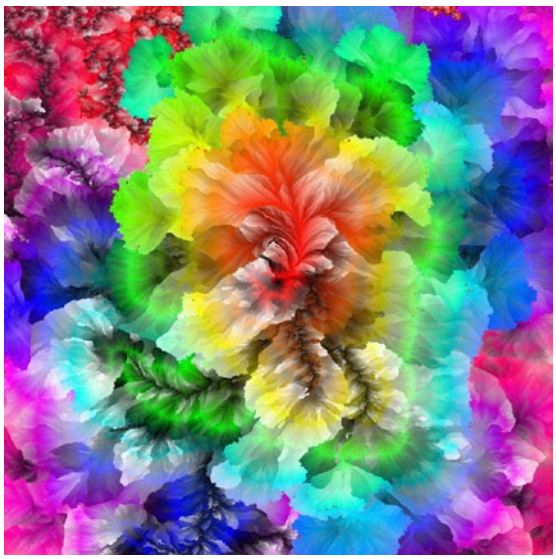
The holy grail: 24 bits

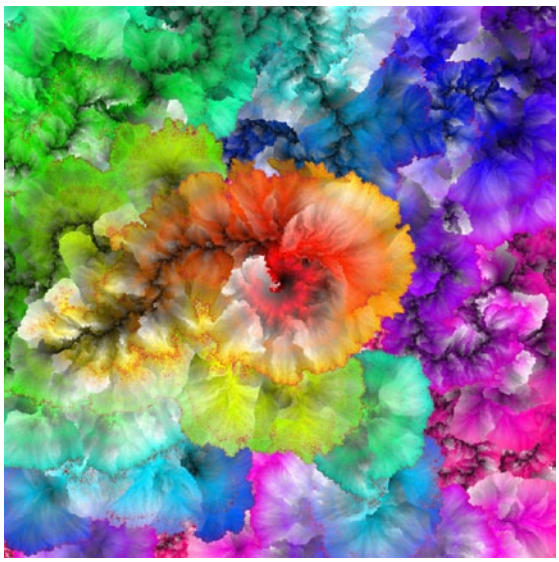
Nowadays, most consumer grade equipment's and software's limit is to display 24 bit colors. So naturally this was my final goal as well. It fits on an image with 4096×4096 resolution (about 16 million pixels).

The optimizations in my software were not finished, but I was too tired and wanted to produce results with what I had, so I ran the renders on a server at my company with CPU time to spare (thank you ArgonSoft). It all took about 50 hours. It would have taken 500 or 5000 hours with a previous version, and maybe it would have taken only 5 hours or 0.5 hours if I had more time, but it doesn't matter now. The images are ready, so let me present you the results. ■









Fejes is an enterprise software developer while at work, and a geek otherwise. He believes that coding is an artform, even without a physical manifestation like these colorful images.

Reprinted with permission of the original author. First appeared in *hn.my/rgb* (joco.name)

Source code of the program can be found here: *hn.my/rgbgen*

Read the follow-up post here: *hn.my/rgb2*

Check out the "Rainbow Smoke" gallery site: *rainbowsmoke.hu*

What I Learned Coding X-Wing vs. TIE Fighter

By PETER LINCROFT

Illustration: Thong Le [weaponix.net]

When we started the X-Wing vs. TIE Fighter project, our goal was to create the first multi-player space combat simulator to be playable over the Internet. There were several major problems that we had to be overcome to accomplish this goal, not the least of which was the Internet itself. I will review the problems we faced, the approach we took, and the results we achieved. I hope the lessons I learned will prove to be valuable to those who read this paper.

The Problems We Knew About

X-Wing vs. TIE Fighter is the third game in the Star Wars space combat simulator series. The Internet was definitely not one of the things that we were thinking about when we created the engine for the original X-Wing game. This was the first problem we faced. Adding Internet capability to an existing engine is significantly more difficult when the engine was not designed with the Internet in mind.

Our second problem was the complexity of the game design. We had always felt that one of the strongest features of our engine was its ability to simulate fairly complex missions. We were proud to have fairly large numbers of craft in each mission, which had reasonably complex behaviors. Our goal in creating X-Wing vs. TIE Fighter was to create a multi-payer game that had this same level of complexity. We wanted to give gamers a multi-player experience that was more complex than "deathmatch." This requirement dramatically increases the amount of data that the players need to have in order to play the game.

Third on our list of problems was that we would not have a dedicated server available; we would have to use a peer-to-peer network model. The expense of providing servers with sufficient processing power and bandwidth for our expected audience size was considered unreasonably high. And because of the nature of the license we were working with, allowing gamers to set up their own servers was not a viable alternative. A peer-to-peer system avoids the problem, but it poses a significantly more challenging engineering problem, because each player must communicate with several other players, instead of with a single server. Because the Internet does not have a viable multi-casting capability, sending the same message to three destinations requires three times as much bandwidth as sending it to a single destination.

The fourth problem, of course, was the Internet itself. When we started the project we assumed that we would need to handle latency that varied from 200ms to a full second. We also knew that we would be limited to the bandwidth available from a 28K modem. These two constraints were our

primary focus when we designed our network model, but they would turn out to be among the easiest problems to solve.

The Approach

Given this set of problems, we designed a network model that we hoped would address all of these issues in a satisfactory way. The first decision we made was the biggest, and would be the source of most of our headaches later on. We decided that we did not want the network model to restrict the complexity of the missions, and we knew that there was no way to compress all of the data relevant to each player into the available bandwidth. We thought of three possible solutions to the problem. The first alternative, and one we knew was being used successfully by other games, was to send only the most "important" data, and allow the rest of the data to be filled in by some form of prediction. The second alternative was to only provide the data necessary to accurately display the world. The third alternative was to send only the actions taken by each player, and simulate the consequences of those actions on each machine.

The first alternative requires the ability to quickly determine which data is "important," and which data is not. In our previous games, players were given a lot of capability to find out what was going on in the game. We even had a real-time map that allowed the player to view all of the craft in the mission simultaneously. In addition, the player could use the "targeting computer" to instantly find out the current status of any craft in the world. If we took the "relevance" approach to the problem, we would have to modify or remove these features.

The second alternative sounded like a good possibility. The typical view from the player cockpit would normally only display a few objects, and if the player could see many objects, the player would be far away and the view would not necessarily need to be completely accurate. The problem was that the player was in open space, flying a very maneuverable craft. They could complete a 360-degree turn very quickly, and in that time they would likely see almost all of the objects in the game. We knew this approach had been successful in games with interior environments, but our game could not use walls to divide the world into manageable chunks. We considered the possibility of introducing a "fog" which would restrict the player's view to only those objects within a certain distance, but let's face it - that's just a bad idea.

The third alternative was immediately attractive to us. The bandwidth required to send only the player's actions would be constant regardless of the complexity of the mission. We had used a similar technique in the past to

allow players to make "recordings" of a game that could be played back in a "VCR" room, so we knew the engine was compatible with the concept. We decided to do a quick test of this approach, and we got our first multiplayer mission working in a matter of days.

Hosting the Game

The second major decision we made was to have one player act as the "host" for the game. Our decision to send the player's input only meant that in a true peer-to-peer system, each player would have to send their messages to every other player. Since there is no broadcast or multicast capability over the Internet, this decision meant that every message would have to be duplicated and sent N-1 times, where N is the number of players in the game. This means that the 28K bandwidth available to the player is really divided by the number of players in the game.

If one player acts as the "host" of the game, we can significantly reduce the burden on the other players, while only slightly increasing the burden on the player that is the host. Each player sends data to the host, who compiles all the data into one large packet, and then sends a copy to each "client." The advantage of this approach is that if the "host" has a faster connection, that person can support a game containing several low-speed players. This did eventually pay off when the game was released, as players with fast connections were able to host eight-player games with the other seven players all playing over modem connections.

The other major advantage of having one player act as the "host" is that we do not have to worry about synchronizing the data on every player's machine with every other machine. Instead, we can focus on every player being synchronized with the game data on the "host" machine. We expected that this would make "late join" easier to implement, but unfortunately that feature never made it into the game.

Despite the ease with which we got our test case working, we did not think our job was going to be easy. We knew this approach would have its own problems. The problem we anticipated with the most anxiety was the one we had seen many times before in our "VCR" feature. When playing back a recording of the player's input, the game would sometimes produce results that were completely different from the original flight. In the past, bugs in the code that were otherwise harmless inevitably caused these "divergence" problems. For example, we might use a local variable as a Boolean flag to decide between two possible actions for a non-player craft. If the variable were accidentally used before being set, the decisions would be random depending on the value of the

variable's location on the stack. This type of bug was usually not noticeable, except when playing back a film. But when playing back a film, the bug could cause a craft to take a different action from the action it had taken when the film was recorded. This difference would quickly "ripple" through the rest of the game world, as craft that were dependent on the actions of that craft made different decisions as well.

If this kind of thing happened during a multi-player game, the players would quickly be experiencing two completely different simulations. We hoped to deal with this "out of sync" problem in two ways. First, we hoped that we would be able to find most of these bugs, and thereby avoid the problem occurring in the first place. Second, we devised a mechanism for detecting when the problem occurred, and "re-syncing" the game by sending the data that had diverged.

The big advantage of this approach was the low bandwidth requirements. We still had to deal with the issue of latency. After some quick tests we realized that even 100ms of latency made our controls unusable. It was incredibly frustrating to try to hit a target when there was a delay between when you pressed the trigger and when your weapons fired. Rather than change the controls and the way the game played to compensate for the problem, we decided to devise a system in which the player would experience nearly zero latency between their actions and the response of their craft. The key to making this work was to use a technique similar to what is sometimes called "dead reckoning".

Our solution was to maintain two simultaneous copies of the game's data. The first copy of the world was based exclusively on the actual actions taken by each player, and was not updated until that information was available. The second copy always represented the state of the game at the current time, and was the version we would render each frame. This second copy of the game data wasn't able to account for the actions of the players because the information about those actions was delayed by the latency of the Internet. Instead, this copy of the game data was based on a prediction of what those players' actions were likely to be. The higher the latency of the connection, the longer the gap of time between the two copies, and the more inaccurate the predicted version became.

Our approach seemed to solve the two Internet problems we had heard the most about: bandwidth and latency. Bandwidth was kept to a bare minimum by only sending data about each player's actions. Latency would cause some inaccuracy in the world (what we called "warping"), but would not affect the player's flight

controls. We were pretty pleased with ourselves, and thought we must be very clever.

Implementing the Design

Our first step was to implement the network model and test it on our LAN. This process went pretty smoothly. Our first implementation was a simple "synchronous" version, in which all the players would wait until all of the input from a frame was received before processing the simulation. This first pass used very little bandwidth, but would not work at all with significant amounts of latency. It also had the significant drawback that if one player had a slow frame rate, all the other players would be slowed down to match the slowest player's frame rate. This was why we called it "synchronous": all of the players were "synchronized" to the slowest frame rate.

This version was fairly easy to code because we did not implement the "predicted" copy of the world, and we did not even try to address the issue of latency. Also, we used DirectPlay, so we have very little work to do to create a game session and get the players joined into it. We got this version up and running quickly so that our mission designers could begin working on multi-player missions. We actually used the "synchronous" version for quite a while. It was good enough to

Illustration: Nige [unusualsuspex.deviantart.com]

test with, so finishing the network code was considered a lower priority than the other issues we needed to address at that stage of development. When we finally came back to the network code we were behind schedule, and that affected some of the decisions we made later in the process. And it meant we were absolutely committed to the complexity of the missions and the user interface.

One big benefit of having implemented this first version early was that we were able to develop some pretty effective techniques for finding "out-of-sync" bugs. Thanks to those techniques and the long period of testing, we actually found most of those bugs. We were also able to work on the "re-sync" mechanism, and we found that on the LAN, we could re-sync a game so quickly that you hardly even noticed when an "out-of-sync" bug had occurred.

When we came back to the network code, we knew the first task was to create a second copy of the world that would be based on the first copy. Unfortunately, our game engine was not coded with this concept in mind, and this turned out to be much more difficult than it should have been. However, once we had the code working, we added some artificial latency to our LAN and tested it out. It worked great!

We now had a version of the game that worked great on the LAN. It used very little bandwidth, and it tolerated 500ms of latency so well you hardly even noticed it. Brimming with confidence, we set up a couple of systems to test it over the Internet. And it worked! We wouldn't realize our mistake until weeks later when we finally did some real testing.

Lessons Learned (The Internet Sucks)

First lesson: If all players dial into the same phone number, you are not testing the Internet. You are testing the modems and the POP server, but you are not testing the Internet. It's obvious when you think about it. Your packets go over the modem to the POP server, and it sends them right back out to the other player. The packets never get past the POP server.

When we finally tried our game on some real network connections, it would fail within seconds. We were mystified. It worked great on the LAN, even with 500ms of artificial latency. When we ran some diagnostics we discovered that we were seeing some simply unbelievable latencies. 5 and 10 seconds was frequent, and we saw some as long as 50 seconds! Our game would simply fall apart under those conditions.

What was actually happening was that a packet would get lost. The TCP protocol specifies that packets will

always be delivered, and furthermore, that they will always be delivered in order. TCP uses a system of acknowledgements to verify that packets are successfully delivered, and will re-send packets if they are lost in transmission. The "in order" specification means that if a packet must be re-sent, the packets that follow it are delayed until the lost packet is received. The problem is that when an Internet connection starts dropping packets, it becomes very likely that the re-sent packet will also get dropped. This means it can take several seconds for a packet to arrive at its destination.

Lesson two: TCP is evil. Don't use TCP for a game. You would rather spend the rest of your life watching Titanic over and over in a theater full of 13 year old girls. First of all, TCP refuses to deliver any of the other packets in the stream while it waits for the next "in order" packet. This is why we would see latencies in the 5-second range. Second of all, if a packet is having a tough time getting to its destination, TCP will actually stop re-sending it! The theory is that if packets are being dropped that it's due to congestion. Therefore, it is worthless to try re-sending because that will only make the congestion worse. So TCP will actually stop sending packets, and start sending occasional little test packets. When the test packets start to get through reliably, TCP will gradually start sending real packets again. This "slow re-start" algorithm explains why we would see latencies in the 50-second range.

Lesson three: Use UDP. The solution to this evil protocol seems simple at first. Don't use TCP, use UDP instead. Unlike TCP, UDP is an unreliable protocol. It does nothing to guarantee that a packet is delivered, and it does nothing to guarantee that a packet is delivered in order. In other words, it does nothing. So if you really need a packet to be delivered, you need to handle the re-sending and acknowledgements. There is one other extremely annoying thing about UDP. Modem connections are made using a protocol called PPP. When you end TCP packets over a PPP connection, it does some very clever compression of the Internet header data, reducing it from 22 bytes to 3 bytes (or less). When you send UDP packets over a PPP connection it does not perform this clever compression and sends the entire 22-byte header over the modem. So if you are using UDP, you shouldn't send small packets.

Of course, our network system absolutely requires that every packet be delivered. If TCP actually worked, this would not be a problem. But TCP is hopelessly broken, so we had to write our own protocol to handle acknowledgements and re-sends. Unfortunately, we didn't realize that right away, and it took us awhile to get there.

Our first step was to switch from TCP to UDP. This was as simple as passing a flag to DirectPlay. Of course, now the game would fail miserably as soon as the first packet was dropped. So, we implemented a simple re-sending mechanism to handle the dropped packets. This seemed to work a little better, but occasionally things would go horribly wrong exactly as they had before. Our first guess was that DirectPlay was actually ignoring the flag and using TCP anyway. But our diagnostics quickly showed us that the problem was even more evil than Microsoft: it was the Internet.

Lesson four: UDP is better than TCP, but it still sucks. We expected packets to be dropped occasionally, but the Internet is much worse than that. It turned out that on some connections, about every fifth packet we sent would just disappear into the Ethernet. When they say UDP is unreliable, they aren't kidding! Our simple re-sending mechanism just didn't perform well enough under these conditions. It was quite common for a re-sent packet to be dropped, and we saw several cases where the original packet and 4 or 5 re-sends of that packet would all be dropped. We were re-sending so many packets, we were starting to exceed the bandwidth of the modem, and then the latency would start to climb, and all hell would break loose.

Our solution was simple and surprisingly effective. Every packet would send a copy of the last packet. This way if a packet were dropped, a copy of it would arrive with the next packet, and we could continue on our merry way. This would require nearly twice as much bandwidth, but fortunately our system required so little bandwidth that this was acceptable. This would only fail if two consecutive packets were dropped, and this seemed unlikely. If it did happen, then we would fall back on the re-sending code.

This seemed to work pretty well! We finally had the game working on the Internet! Sure the Internet had turned out to be far worse than we had thought, but we could deal with it.

Lesson five: Whenever you think the Internet can't get any worse, it gets worse. More extensive testing showed that we still had some serious problems. Apparently we had some kind of bug in our re-sending code, because it seemed that occasionally players would just lose their connection and nothing would get through. After spending endless hours trying to find the bug in our code, we finally realized that our code was fine, it was the Internet that was broken!

It turns out that sometimes the Internet gets so bad, that practically no packets get through at all! We documented periods of 10 and even 20 seconds during which only 3 or 4 packets

would be delivered. No wonder TCP decides to just give up! How can you possibly play a game under conditions like that? We had a major problem on our hands. This "lost connection" phenomenon was something we just weren't prepared to deal with.

Fortunately, this condition is usually pretty short, on the order of a few seconds. We managed to get our code to handle that by just tweaking the re-sending code. The player who is suffering this condition will frequently have their game stopped while we wait for the connection to clear, but once the condition passes, they can resume playing.

Unfortunately, this "lost connection" condition can last pretty long, and when that happens, we just can't handle it, and we end up having to disconnect that player from the game. This isn't really a solution, but at least it meant one bad connection wouldn't ruin everyone's game.

One of the last refinements we made to the game to deal with the Internet involved dealing with the inaccuracy of the predicted world. Since latencies could be very long, we need a way to deal with the inaccuracy of the predicted world.

Our first clue that we had to address this issue was the result of implementing what we thought would be an improvement. We realized that if any one player had trouble getting their data to the host computer, then every player would suffer because the host would not send out the compiled data packets until it had received data from every player. We decided that if a player failed to get their data to the host within a reasonable amount of time, then we would simply drop that data and send out the compiled packet without it.

If you follow through the consequences of that action you will realize that it creates a very evil situation. Players normally predict the position of their own craft with perfect accuracy. After all, they know exactly what they have done, so they know exactly where they should be. But if the host drops their input from the "official" version of the world which is the basis of their predicted version of the world, then they will actually have to change their own position if they are going to stay in sync with the other players. The visual result of changing the local player's position is that the position of everything in the world, including the star-field, will change position.

This effect, dubbed "star-field warping," is extremely disconcerting, and makes the game practically unplayable. We eventually compromised by only dropping a player's data if it was extremely late, which made this event fairly rare. However, in hindsight it might have been better to use the same solution we eventually implemented for the other players.

This instantaneous jump in position, or "warp", will always occur for the other players, since their position is always incorrectly predicted. If latency is fairly low (less than 200ms) this jumping is not very noticeable, but as latency increases, the inaccuracy of the predicted world increases, and this "warping" effect becomes more noticeable.

To address this problem, we implemented a "smoothing" effect. The smoothing algorithm keeps track of our last prediction of each player's position. It then takes the current prediction and moves it closer to the last prediction. This effectively smoothes out the motion of the other player's craft, and it looks much better, even though it is probably less accurate.

Conclusions Drawn

The conclusion is obvious: the Internet sucks. We were pretty disappointed in how our game performed over bad Internet connections. But looking back on it now, I believe we did as good a job as anyone else, given the style of game we were building, and the constraints we were forced to deal with.

The lack of a dedicated server turned out to be a huge problem. In cases where the "lost connection" phenomenon lasted more than a few seconds, it was clearly easier to send the entire state of the world than it was to resend all the packets that had been lost. This was not practical, however, because the computer that would have to do that would be one of the players', and could not spare the bandwidth. A dedicated server could have addressed this problem, and doing so would have been equivalent to allowing a player to "join" a game that was already in progress. "Late join" was a feature we really wanted to have in the game, but we felt it just wasn't practical without a dedicated server.

A dedicated server would also have made it easier to support more simultaneous players. The latency would be cut nearly in half, because messages would not have to go through modems before being re-sent to the other players, as they do with the "host" player. In addition, a dedicated server would make it significantly easier for a player to evaluate the quality of their connection to the game, since they would only have to worry about their connection to the server. With a player acting as a host, the other players must be concerned with the quality of the host's connection to the Internet, as well as their connection.

One of the biggest problems we faced with our network model was the requirement that packets be processed in order. Out-of-order packets could be used to improve the predicted copy of the world, but in XVT they are not. Even if they were, there would still be a significant performance problem.

The problem is that when the in-order packet finally does arrive, we must process it, and all the out-of-order packets that have come since. This can be time-consuming because the simulation must be run on each packet.

Both of these problems would have been much easier to address if we had started from scratch. But because we were modifying an existing engine we were limited by its capabilities. If the engine had been able to simulate large time steps more efficiently, that would have helped a great deal. We were effectively required to use a fixed time step, and this made simulating a long time step very inefficient. In addition, if the engine had been able to use out-of-order data to improve the predictions, then the long lag for a re-sent packet would have been much less noticeable.

One of the advantages of our approach to the problem is that it is pretty much completely independent of the game's content. The packets we send only contain data about the player's input device, and this technique could work virtually unchanged for almost any kind of real-time game. The really nice thing about this aspect of the model is that we did not have to worry about changing the content of the game, requiring us to change the network code. The fact that no game-specific data is included in the packets also makes it much more difficult for players to cheat by using "bots". In order to give an advantage, a "bot" would have to be able to create a stream of input data that is more effective than a human player and I think this would be extremely difficult. ■

After earning his degree in Computer Science from the University of California, Peter went into the games industry, passing quickly through a failed start up and a short stint at LucasFilm Games before landing at Totally Games. At TG, he was fortunate enough to become the lead programmer of the X-Wing and TIE Fighter series of space combat simulators, fulfilling a childhood dream to make a top quality game in the Star Wars universe. After starting a family, he left the games industry and have since pursued a successful career in high tech startup companies, most notably BigFix, Inc., and currently Tanium, Inc.

This paper was originally published in the 1999 Game Developer's Conference proceedings.

Startup Sales Negotiations 101

How to Respond to Discount Inquiries

By STELI EFTI

People will sometimes reach out and ask for a discount on your product before they take the time to sign up for a trial and use it at all. What do you do when that happens?

Instead of debating if you should or shouldn't offer them a discount right away, you need to refocus their energy on what really matters: your product!

Let's explore the 3 core reasons why you never want to negotiate pricing before someone had a chance to trial your product and determine that it's a good fit.

1 You're starting the relationship on the wrong foot

People who ask you to lower your prices before having invested any time using your product are usually trouble.

This can often lead to winning a new customer that is going to expect you to give 24/7 premium phone support and prioritize features based on their needs all while trying to pay you pennies on the dollar. If you start the relationship by giving them everything they ask for, don't be surprised if they keep asking for more in an unreasonable fashion. This is ultimately unsustainable and unhealthy for both sides.

2 They're buying for the wrong reason

At this point they can't tell if your product is a good fit for them since they never used it. Your first priority should always be to help people explore and discover that your product is really solving their problem before negotiating what the final pricing should be.

Discounting your product upfront might help you close some deals faster but will often lead to these customers ultimately discovering that they should have never bought in the first place.

Always be wary of prospects that don't want to do their homework upfront. Nothing sucks more than a new customer that cancels immediately after having created a ton of support and onboarding cost.

3 You're negotiating on price vs. value

The problem with people trying to negotiate pricing before testing your product is that you are forced to negotiate on price rather than value.

They didn't have a chance to build up any desire to buy and discover the massive value your product could deliver to them. All of the sudden your product turns into a commodity and your only differentiation is offering them the lowest price possible.

4 You're negotiating without leverage

The more time people invest in your product the more "invested" they become and naturally the harder it is for them to "throw away" the time they put into exploring your product and making it part of their daily workflow.

You always want to postpone the most difficult/complex parts of the sales negotiation till the end of the sales cycle. That way you ensure the right amount of momentum as you move forward in the sales process and avoid too much upfront friction.

Here is what your response should be when someone asks for a discount without having tried your product:

"Thanks for inquiring about pricing options! Why don't you sign up for a trial and give the product a go? If you find out that it's a great fit I'll take care of you and make sure you get a price that makes you happy. Sound fair enough?"

This works every time. The reply you usually get will be:

"Great! Just signed up and giving the product a go. Thanks!"

What's the result you should expect?

9 out of 10 times the people that turn out to be a bad fit will self-select during a trial and just leave. The prospects that are a good fit will love your product so much that they will not negotiate hard for a discount since they now really understand its value.

Even if they do, it's fine to give great customers a good price because you know they are buying for all the right reasons and will probably stay with you for a long time.

We've done this thousands of times and it always works. I hope this startup sales negotiation tactic serves your business as much as it has ours. ■

Steli Efti is the Co-Founder and CEO of *Close.io*, a sales communication software that empowers startups to make more sales and close more deals.

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

How We Made Trello Boards Load Extremely Fast In A Week

By CSABA OKRONA

We made a promise with Trello: you can see your entire project in a single glance. That means we can show you all of your cards so you can easily see things like who is doing what, where a task is in the process, and so forth, just by scrolling.

You all make lots of cards. But when the site went to load all of your hundreds and thousands of cards at once, boards were loading pretty slowly. Okay, not just pretty slow, painfully slow. If you had a thousand or so cards, it would take seven to eight seconds to completely render. In that time, the browser was totally locked up. You couldn't click anything. You couldn't scroll. You just had to sit there.

With the big redesign, one of our goals was to make switching boards really easy. We like to think that we achieved that goal. But when the browser locked up every time you switched boards, it was an awfully slow experience. Who cared if the experience was easy? We had to make it fast.

So I set out on a mission: using a 906 card board on a 1400×1000 pixel window, I wanted to improve board rendering performance by 10% every day for a week. It was bold. It was crazy. Somebody might have said it was impossible. But I proved that theoretical person wrong. We more than achieved that goal. We got perceived rendering time for our big board down to one second.

Naturally, I kept track of my daily progress and implementation details in Trello. Here's the log.

Monday (7.2 seconds down to 6.7 seconds. 7% reduction.)

Heavy styles like borders, shadows, and gradients can really slow down a browser. So the first thing we tried was removing things like borders on avatars, card borders, backgrounds and borders on card badges, shadows on lists, and the like. It made a big impact, especially for scrolling. We didn't set out for a flat design. Our primary objective was to make things faster, but the result was a cleaner, simpler look.

Tuesday (6.7 seconds down to 5.9 seconds. 12% reduction.)

On the client, we use Backbone.js to structure our app. With Backbone.js, it's really convenient to use views. Really, very convenient. For every card, we gave each member its own view. When you clicked on a member on a card, it came up with a mini-profile and a menu with an option to remove them from the card. All those extra views generated a lot of useless crap for the browser and used up a bunch of time.

So instead of using views for members, we now just render the avatars and use a generic click handler that looks for a `data-idmem` attribute on the element. That's used to look up the member model to generate the menu view, but only when it's needed. That made a difference.

I also gutted more CSS.

Wednesday (5.9 seconds… to 5.9 seconds. 0% reduction.)

I tried using the browser's native `innerHtml` and `getElementByClassName` API methods instead of jQuery's `html` and `append`. I thought native APIs might be easier for the browser to optimize and what I read confirmed that. But for whatever reason, it didn't make much of a difference for Trello.

The rest of the day was a waste. I didn't make much progress.

Thursday (5.9 seconds down to 960ms)

Thursday was a breakthrough. I tried two major things: preventing layout thrashing and progressive rendering. They both made a huge difference.

Preventing layout thrashing

First, layout thrashing. The browser does two major things when rendering HTML: **layouts**, which are calculations to determine the dimensions and position of the element, and **paints**, which make the pixels show up in the right spot with the correct color. Basically. We cut out some of the paints when we removed the heavy styles. There were fewer borders, backgrounds, and other pixels that the browser had to deal with. But we still had an issue with layouts.

Rendering a single card used to work like this. The card basics like the white card frame and card name were inserted into the DOM. Then we

inserted the labels, then the members, then the badges, and so on. We did it this way because of another Trello promise: real-time updates. We needed a way to atomically render a section of a card when something changed. For example, when a member was added it triggered the `cardView.renderMembers` method so that it only rendered the members and didn't need to re-render the whole card and cause an annoying flash.

Instead of building all the HTML upfront, inserting it into the DOM, and triggering a layout just once; we built some HTML, inserted it into the DOM, triggered a layout, built more HTML, inserted it into the DOM, triggered a layout, built more HTML, and so on. Multiple insertions for each card. Times a thousand. That's a lot of layouts. Now we render those sections before inserting the card into the DOM, which prevents a bunch of layouts and speeds things up.

In the old way, the card view render function looked something like this…

```
render: ->
  data = model.toJSON()

  @$.innerHTML = templates.fill(
    'card_in_list',
    data
  ) # add stuff to the DOM, layout

  @renderMembers()
  @renderLabels()
  # add even more stuff to the DOM, layout
  @
```

With the change, the render function looks something like this…

```
render: ->
  data = model.toJSON()
  data.memberData = []

  for member in members
    memberData.push member.toJSON()

  data.labelData = []
  for labels in labels when label.isActive
    labelData.push label

  partials =
    "member": templates.member
    "label": templates.label

  @$.innerHTML = templates.fill(
    'card_in_list',
    data,
    partials
  ) # only add stuff to the DOM once, only
one
    # layout
  @
```

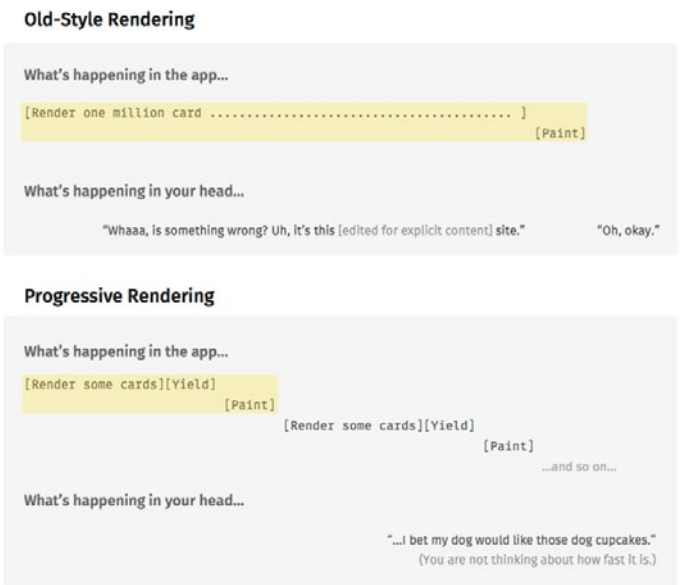
We had more layout problems, though. In the past, the width of the list would adjust to your screen size. So if you had three lists, it would try to fill up as much as the screen as possible. It was a subtle effect. The problem was that when the adjustment happened, the layout of every list and every card would need to be changed, causing major layout thrashing. And it triggered often: when you toggled the sidebar, added a list, resized the window, or what not. We tried having lists be a fixed width so we didn't have to do all the calculations and layouts. It worked well so we kept it. You don't get the adjustments, but it was a trade-off we were willing to make.

Progressive rendering

Even with all the progress, the browser was still locking up for five seconds. That was unacceptable, even though I technically reached my goal. According to Chrome DevTools' Timeline, most of the time was being spent in scripts. Trello developer Brett Kiefer had fixed a previous UI lockup by deferring the initialization of jQuery UI droppables until after the board had been painted using the queue method in the async library. In that case, "click … long task … paint" became "click … paint … long task."

I wondered if a similar technique could be used for rendering cards progressively. Instead of spending all of the browser's time generating one huge amount of DOM to insert, we could generate a small amount of DOM, insert it, generate another small amount, insert it, and so forth, so that the browser could free up the UI thread, paint something quickly, and prevent locking up. This really did the trick. Perceived rendering went down to 960ms on my 1,000 card board.

That looks something like this…



Here's how the code works. Cards in a list are contained in a Backbone collection. That collection has its own view. The card collection view render method with the queuing technique looks like this, roughly…

```
renderQueue = new async.queue
(models, next) =>
   @appendSubviews(@
subview(CardView, model) for model
in models)
   # _.defer aka setTimeout(fn,
0), will yield
   # the UI thread so the browser
can paint.
   _.defer next
 , 1
 chunkSize = 30
 models = @getModels()
 modelChunks = []
 while models.length > 0
   modelChunks.push(models.
splice(0, chunkSize))

for models in modelChunks
   # async.queue flattens arrays so
lets wrap
   # this array so it's an array
on the other end
   renderQueue.push [models]
```

We could probably just do a `for` loop with a `setTimeout 0` and get the same effect since we know the size of the array. But it worked, so I was happy. There is still some slowness as the cards finish rendering on really big boards, but compared to total browser lock-up, we'll accept that trade-off.

We also used the `translateZ: 0` hack for a bit of gain. With covers, stickers, and member avatars, cards can have a lot of images. In your CSS, if you apply `translateZ: 0` to the image element, you trick the browser into using the GPU to paint it. That frees up the CPU to do one of the many other things it needs to do. This browser behavior could change any day which makes it a hack, but hey, it worked.

Friday

I made a lot of bugs that week, so I fixed them on Friday.

That was the whole week. If rendering on your web client is slow, look for excessive paints and layouts. I highly recommend using Chrome DevTool's Timeline to help you find trouble areas. If you're in a situation where you need to render a lot of things at once, look into async.queue or some other progressive rendering. ■

Bobby Grace is a designer and developer working at Fog Creek Software in New York City. He is big time into computers and eating and bouldering and raw juice.

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

HACK ON YOUR SEARCH ENGINE

and help change the future of search

duckduckhack.com

Confessions of an Intermediate Programmer

By MICHAEL BROMLEY

I AM AN INTERMEDIATE programmer.

I have a pretty good grasp of the basics. I have made enough mistakes to have a good idea why they were mistakes. I am aware there is a lot that I need to know more about. Crucially, I have some idea of what those things are, and I am actively and energetically working on improving.

It has taken a while for me to get to the point where I am confident enough to admit that I am only average in ability. I no longer feel the need to hold second-hand opinions that I don't really understand. I'm not so afraid of being found out when I don't know about something.

It hasn't always been this way. You might not credit it, but I used to be something of a programming guru.

This erroneous evaluation of my own ability can best be attributed to the relatively isolated environment in which I developed my skills. Back in those days, even owning a computer was a little bit special; knowing how to use it even more so.

By my own estimation, I was a pretty knowledgeable and experienced programmer. By the time I was barely out of my teens, I'd written programs in C++, Pascal, C#, JavaScript and — my crowning glory — I had written a custom e-commerce platform in PHP from scratch (more on this later).

In reality, I was perhaps just a few cuts above that "friend's son who is a whizz with websites!" I had had no interaction with any other programmers, so my only point of comparison was the people around me; people who either didn't bother much with computers, or if they did they probably had five spammy toolbars clogging up their

Internet Explorer window. People who might well use the phrase "my Internet is broken."

Here is the story of how I fooled myself into thinking I was much better than I was.

The Genesis of My Genius

When I was about nine years old, a friend of mine had satellite TV at his house. At home, we were limited to the standard four UK terrestrial channels (these were the days before Channel Five - how did we manage?), and I hankered after the overwhelming choice of bad TV that I had just witnessed. All we needed was one of those satellite dishes — or "satellites" as we called them — and I, too, would be able to watch QVC or Eurosport whenever I wanted. Somehow dimly aware of my nascent gift, I set about to build my own satellite (dish)!

My design involved a fully opened umbrella and a length of copper audio cable, one end attached to the metal shaft of the umbrella, the other stuffed into my TV's aerial socket. Admittedly, my design had some flaws, and consequently failed to deliver the expected results. However, the point of this anecdote is simply to demonstrate the technical ambition that would mark my childhood and adolescence. Nobody else I knew had even thought about making a satellite.

A few years later, I became an early-adopter of the Internet when my dad got a 14.4k modem at his office. I recall spending one Saturday afternoon patiently waiting for the flaming Manga logo gif to load, each subsequent frame appearing every minute or so. I even built my own website using Netscape Composer. Not yet aware of the architecture of the Internet, I saved my html files locally and then wondered when they would show up online. This detail, however, did not detract from the fact that nobody else I knew had made their own website.

By the time I reached my early teens, I discovered the darker side of my talent. Armed with a copy of the Jolly Rogers Cookbook, a couple of friends and I set about to shake the technological (and moral) foundations on which mid-90s England stood. Phreaking was our forte. We got as far as using a handheld acoustic coupler to make free international calls from public phones to American girls we'd met on ICQ and setting up voicemail boxes on private branch exchanges. Schoolwork and skateboarding prevented us from taking our exploits much further. Had we not such distractions, we'd have no doubt been regularly making napalm, hacking government networks and killing men with our bare hands. Although we failed to fully explore the limits of our powers, the fact was nobody else but us owned an acoustic coupler.

Despite my numerous adventures and misadventures with various technologies thus far, something was still lacking. My ideas were always several steps beyond my physical abilities — as highlighted by the "satellite" episode. I needed a way to get the contents of my mind out into the world. I needed a direct interface between my imaginings and reality.

The Fuck Generator

The true turning point came when I was about fourteen years old. I bought a copy of PC Plus magazine which included a cover CD featuring a full version of Borland C++ Builder. I installed it and carefully followed the "hello world" tutorial which was helpfully included in the magazine.

This was it. A new world opened up before me. The restrictions imposed upon my imagination by the material world were gone. My creativity unshackled, the cathedrals in my mind would be made manifest! To what lofty end should I put this new-found tool? It was obvious. The Fuck Generator.

As simple as it was elegant, the Fuck Generator (fgen.exe) was a command-line program, and my first advance beyond "hello world." Upon starting, it would prompt the user for a number. With this number n, it would then print out the string "fuck," n times. Finally the user was given the option to repeat the exercise, or quit. Perhaps a little limited in use, I nevertheless was hooked on the power that I had tasted. It is a particular joy that any programmer will know well, to see the machine do your bidding, no matter how simple a task that may be. It works, and it works because you understand how to make it work. And it cannot do anything but work.

A short while later, another edition of PC Plus included a full version of Borland Delphi. With it, I upgraded the concept to include a Windows GUI and the ability to randomly generate colourful and sometimes surprising 4-part insults. While the other kids at school were passively playing PlayStation, I was engaged upon a far more meaningful and creative endeavour. I was generating fucks.

By this point, it was quite clear that I was destined for big things. It was time to show the world what I could really do.

My Magnum Opus

In the late 90s, I created a website for a small but expanding mail-order retailer. At first, the site was just a few static pages — brochure-ware — complete with a navigation menu in a frameset and the obligatory visitor counter on the home page.

When we started getting more and more enquiries from the website, we decided to experiment with adding e-commerce functionality. We iterated

over several off-the-shelf packages, whose quality ranged from utterly terrible to just terrible. My memory of the first version is predominated by fiddling about with cgi scripts and the bizarre use of <select> elements for almost all user interaction. A later version was a monstrosity of framesets and JavaScript — long before it was anywhere near advisable to base your app's functionality on JavaScript. Another version was powered by a Microsoft Access database.

At length we came to the realisation that, if we wanted to have a genuinely okay-ish or even decent online shop, we'd need a custom solution. I considered my past success with fgen.exe and its sequel, not to mention a string of excellent websites I'd built by this time, case in point: my Manic Street Preachers guitar tab archive website was pretty authoritative, and a proud member of the "Manics Web Ring" (remember web rings?). I felt the time had come to really see what I was capable of. I'd build it myself. From scratch.

From scratch?! If open-source frameworks existed at that time, I didn't know about them. No — I had my own plan. I bought a book on PHP and MySQL, and started to learn both technologies as I built the new website.

As luck would have it, the book featured as one of its central examples a very simple shopping cart application. All the parts were there — "category.php" would list all the products in a category; "product.php" would display the details of a product with a button to add it to the cart; and most importantly, "cart.php", where the real magic would happen. This was clearly meant to be!

I followed the example studiously, faithfully implementing all the ingenious and no-doubt cutting-edge techniques — those handy "mysql_" functions for data access; string concatenation for building queries; separating functions into a "functions.php" file; including a "header.php" and a "footer.php" to maintain consistency site-wide; shunning the bulky overhead of the object-oriented approach (whatever that really meant) in favour of lightening-fast procedural code. My skills were increasing exponentially!

Like a one-man termite colony, I built towers and dug labyrinthine tunnels of code. The structure stretched both further skywards and deeper underground with each new feature that I added. And add features I did. Customer accounts, product ratings, order histories, reward points, voucher codes, special offers, logging, A/B testing, encryption of payment data, and on and on. A sprawling maze of interconnected dependencies, a galaxy of functions of all shapes and sizes, slowly spinning around a central, immovable hub: "cart.php."

After about eight months of feverish work, it was finally ready.

Now, my knowing reader, you may be expecting me to detail how spectacularly, horribly wrong it all went once we flipped the switch on our new website. I am afraid I have to disappoint you.

It worked.

Worst Practices

Despite what I now refer to as my "worst practices" approach, the thing worked. Every bad tutorial, every anti-PHP blog post — it was all there. Spaghetti code? Check. Inconsistent naming of data and routines? Check. Presentation mixed — nay, fused — with business logic? Check. Magic numbers and global data galore? Check.

To me, the object-oriented approach was just a bunch of unnecessary overhead and boilerplate, and I had plenty of misinformation to back me up. I knew all about testing too — click through your feature a few times, seems good, upload to production! I was dimly aware of other (fancy, overly-complex) architectures but as far as I was concerned, mine was a perfectly sensible (and probably much faster) way of doing things.

The proof of my rightness in all these things was the fact that I had written, from scratch, with my bare hands and wits, a functioning and full-featured e-commerce website. Furthermore, one that performed well and was successful and expanding!

In my eyes, there was not much difference between me and the guys who wrote Amazon.com. Sure, Amazon was quite a bit bigger, but I saw no reason why my platform would not scale up without a problem — especially considering the blazingly fast procedural architecture I had used.

And so I had reached a plateau of skill as a programmer. That's not to say I was disinterested in learning more — I just didn't feel an urgency about doing so. After all, I had built something that worked and worked well. Surely anything beyond that was just a bonus, the cherry on top.

Back Down To Earth

This state of affairs prevailed, I'm sorry to say, for several years. I was only working on the site on a very part-time basis, spending the majority of my days working in a completely different field. Over the years of maintenance and the occasional adding of new features, I did develop an awareness that certain choices I had made were now proving to be bothersome. I noticed how long it sometimes took to find what I was looking for in the source files. I was perturbed by the number of minor bugs that would emerge in seemingly unrelated areas of the site each time I made a change.

My learning did not completely stagnate, but it did crawl along pretty slowly. For example, I came to learn that the mysql_ functions that I had used were now considered risky, to the degree that support would be dropped in future versions of PHP! For a long time, I countered any fears with the knowledge that my water-tight sanitization routines would more than make up for that. After all, I had tried various SQL-injection strings in pretty much every form input I could find, and it all seemed hunky-dory.

One day last year I got an urgent call — the website was down! Every request resulted in a 500 internal server error! After the engineer at the hosting company had got it up and running again and had conducted the post-mortem, it turned out we had been the victim of an exotic SQL injection attack, the like of which I had never seen before (in any of the several tutorials I'd read about SQL injection).

Alright, I thought, maybe it's time to swap over to this new-fangled PDO thing I've been hearing about.

My Epiphany

When I sat down to re-write all the data-access functions, something profound occurred. I realised that this was going to be tough. And I realised why it was going to be tough.

It was going to be tough because these functions were scattered all over the place; because I had no real way of knowing if I was going to break something in some subtle way; because the code was inconsistent and I'd have to carefully study how each instance slightly differed from the last; because much of the code was tightly coupled with other parts which might also subtly break when I made changes. In short, it was going to be tough because of all the bad practices and lack of understanding that had informed the creation of this sprawling mess that only now revealed itself to me.

All the justifications, the defensive reasoning, the ignorance started to melt away. I had been wrong. I was not the sublimely gifted programmer I had suspected myself to be. I was a fake who had somehow gotten away with it for this long!

My folly had been thrown into sharp relief, and though this was a blow to my ego, it was also an incredibly valuable lesson. I learned first-hand — and painfully — why there is a right way and a wrong way to do things. It's not just a matter of taste or fad. It's not a matter of who has the cleverest arguments. The right way has real-world ramifications which will make your life (and the lives of others who touch your code) better. The wrong way leads to frustration and wasted time. I won't try to address here the thorny issue

of what exactly constitutes "the right way." Suffice it to say it wasn't what I had been doing.

The Real Sin

I did implement PDO. At the same time, I started using PHPUnit for the first time. Attempting to retro-fit that kind of code with unit tests is not something I would like to repeat.

Nowadays I make a conscious effort to push myself and learn more whenever I can. I am reading the books that programmers are supposed to have read. I'm following blogs. I'm listening to podcasts. I'm watching conference videos. I'm attending and even giving talks at local user groups. I'm working on side-projects to challenge myself to learn brand new technologies. I'm trying to learn the right way to do things.

For all you who are also engaged upon this task, there is an important factor in our favour. Being that programming is such an utterly abstract endeavour, the material-world limitations that characterize so many other fields simply do not apply. Here, the limiting factor is oneself.

I'll close this story with some true words of wisdom. At the time I began drafting this blog post, I was just finishing the book Code Complete Second Edition by Steve McConnell. Towards the very end of the book, at the bottom of page 825, he writes something that perfectly articulates the exact sentiment I had in mind when writing this post. Perhaps it is telling that he was able communicate in two sentences what took me a couple of thousand words:

> *"It's no sin to be a beginner or an intermediate. It's no sin to be a competent programmer instead of a leader. The sin is in how long you remain a beginner or an intermediate after you know what you have to do to improve."*

Michael Bromley is a reformed programming guru, now learning how to write software properly. He lives in Vienna with his wife and son. You can reach him via *@michlbrmly*

Reprinted with permission of the original author.
First appeared in *hn.my/confession* (michaelbromley.co.uk)

The Best Programming Language

How to Stop Worrying and Love the Code

By ÁLVARO CASTRO-CASTILLA

Every once in a while, someone, somewhere, decides it's time to write yet another post on what the best programming language is, the mighty properties of a forgotten language, or the new language that does it right. So my time has come. Finally, I get to say what I think about programming languages.

First of all, a disclaimer: unless you've developed in 30+ languages, and suffered the code of others in all (or most) of them, you can't really be objective. So yes, I'm biased. Like most of the people writing about this topic. Actually, I believe that this topic becomes absurd as soon as you are well-versed in many languages.

TL;DR: The Great Languages

I hereby declare these languages to be The Great Languages within the realms of my blog.

- **Assembly**: the language of the machine.
- **C**: the systems language.
- **JavaScript**: the language of the web.
- **Scheme**: the lightweight, embeddable and extremely flexible language that compiles to both C and JavaScript.

Most of the code examples are from *RosettaCode.org*

Ada

I've always been curious about the idea of designing a language around memory safety. That makes sense for applications in real-time operating systems and critical systems in general. Probably if you are considering using this language you don't need to read this, and you come from a highly specialized background. This is one of the languages that you use once you know what you are doing, and then you don't have many options either. Some bits of Ada:

```
function Best_Shuffle(S: String)
return String is
      T: String(S'Range) := S;
      Tmp: Character;
   begin
      for I in S'Range loop
         for J in S'Range loop
            if I /= J and S(I) /=
T(J) and S(J) /= T(I) then
               Tmp  := T(I);
               T(I) := T(J);
               T(J) := Tmp;
            end if;
         end loop;
      end loop;
      return T;
   end Best_Shuffle;
```

It looks safe, right? ;)

Bourne (Again) Shell

I always think: do I really need to write this Linux script in a shell language? Is it really necessary? It doesn't matter if you don't write your scripts in shell, because eventually you'll deal with one of these scripts face to face, and you'll wonder how they did it in the Bare Metal Age, pre-stackoverflow. Anyway, with the right book, you'll start thinking that the language just needs some make-up (and consistency). There is nothing amazing about this language, nothing that will expand your mind or make you more productive, or that justifies it from the business point of view. It is just pervasive in the *nix world. Nevertheless, it's a must for system administration, and it isn't as bad as it looks. It's a bit like JavaScript, you need to know the good practices more than with other languages.

When would I use Unix Shell?

- OSX/Linux/POSIX system administration
- For task automatization
- To unlock command-line superpowers

Some Bourne Shell code. Enjoy those boolean expressions!

```
#!/usr/bin/env sh

l="1"
while [ "$l" -le 5 ]
  do
  m="1"
  while [ "$m" -le "$l" ]
    do
    printf "*"
    m=`expr "$m" + 1`
  done
  echo
  l=`expr "$l" + 1`
done
```

C

Well, you have to respect C, even if you don't like it. It's arguably one of The Great Languages. The language that programs machine reality (not models of). It's the father of UNIX, all the languages of the capital C, and the lingua franca of systems development. It's been battle-tested, time-tested and hype-tested. The plethora of tools available for developing, debugging, profiling and supporting C development make it for all its defects as a language (not so many, in my opinion). It's a language that really achieved its purpose: become a general-purpose Assembly language for every processor. Nowadays, it is the de-facto Assembly for even the strangest architecture, and it has become very hard to make better hand-crafted code than that generated by C compilers.

It is thus a powerful tool, but one that needs to be mastered. The language shows no mercy, and you always need to know what you are doing. That is what makes C the language for understanding the machine. There is beauty in this, and there is a practical side too: there are things that just can't be done without the kind of low-level that C provides. C programmers must understand very well what they are doing, leading to very solid software in the long run. If there is something that could debunk C is a low-level language with great support for concurrency. Or maybe a mythical language with the properties of Haskell and the pervasiveness of C.

C++

A monster. It was my first language, and I didn't really understand how it was screwing my productivity and limiting my skills until I tried many others. The bad reputation of C++ is promoted by some well-known programmers, and I agree completely. C++ seems as if Bjarne Stoustrup took every single feature he could think of and added it to C. The cognitive load it imposes might make you more than 80% less productive. Think of it this way: you have a brain of X capacity, and that capacity is limited, doesn't matter how much capacity you have, and you want to leave as much as possible of it for the important things. The wise thing to do would be to reduce the amount of brain power used for the language per se, and use the most of that brain for solving the problem and encoding an algorithm. If the language is complex, no matter how smart you are, you'll need to use more of your brain for the syntax and the semantics of the language and less to efficiently projecting your ideas onto code.

I think C++ is the quintessential example of too much complexity for not that much gain. I agree, building large programs in C is difficult (but arguably an option, look at the Linux Kernel). Go, Rust and D are better languages by all measures, except for the fact that C++ is what the world actually uses.

C#

Enterprise language that aims at reducing any kind of programmer creativity that might hinder its replaceability in any large organization. Object-oriented, statically typed, verbose, with heavy libraries and lots of boilerplate. You can see Microsoft's hand behind this creation. But don't get me wrong, it is not a bad language. It just isn't sexy, which precisely is what Microsoft wanted in the first place. At least, it is a radical improvement when compared with Visual Basic. I would use it for:

- Windows development.
- Game development (well, mostly

because Microsoft forces developers, but I would still prefer good ol' C/C++).

- There are huge things going on in this language: Unity3D, Xamarin, .NET, XNA.

Objective-C

I have a much better opinion of Objective-C than of C++ (and C#). It's syntactically ugly, but I like it as a language. It's got a great set of libraries based on NextStep, with the plus of being a real improvement upon C, without growing too much out of control and bringing ambiguities in keywords with its parent language. As I said, it's a bit ugly and difficult to read, especially when nesting functions, but definitely its beauty resides in its conceptual approach, not in its syntax. See this nested call:

```
char bytes[] = "some data";
NSString *string = [[NSString
alloc] initWithBytes:bytes
length:9 encoding:NSASCIIStringEn
coding];
```

This is beautiful code for a son of C language, making use of Objective-C's so-called blocks.

```
#import <Foundation/Foundation.h>

typedef NSArray *(^SOfN)(id);

SOfN s_of_n_creator(int n) {
  NSMutableArray *sample
= [[NSMutableArray alloc]
initWithCapacity:n];
  __block int i = 0;
  return ^(id item) {
    i++;
    if (i <= n) {
      [sample addObject:item];
    } else if (rand() % i < n) {
      sample[rand() % n] = item;
    }
    return sample;
  };
}

int main(int argc, const char
*argv[]) {
  @autoreleasepool {

NSCountedSet *bin = [[NSCountedSet
alloc] init];
    for (int trial = 0; trial <
100000; trial++) {
      SOfN s_of_n = s_of_n_cre-
ator(3);
      NSArray *sample;
      for (int i = 0; i < 10; i++)
{
        sample = s_of_n(@(i));
      }
 [bin addObjectsFromArray:sample];
    }
    NSLog(@"%@", bin);

  }
  return 0;
}
```

Clojure

Being a Scheme programmer I have respect for Clojure: it's a so-called modern Lisp, with some unique features. I'd say Clojure's strong points are Java interoperability and concurrency utilities in the core language. It's a sibling of Scala, but differs in their flavor: lisp vs. hybrid OOP/functional, making Clojure less popular due to the excess of parentheses. Choosing one of these two for a project is a matter of taste, because neither are proven technologies with a long track record of successful production applications, as compared with Java or PHP, although they both stand on the shoulders of JVM. Another thing to take into consideration for any JVM-base language is the startup time of the virtual machine: it doesn't seem like a very lightweight solution for small tasks. These are the situations where I would use Clojure:

- Web development. There are good options for this, and the Clojure community seems very active in this area.
- When you want to use the JVM technology without the Java thing. Both programmer happiness and productivity will improve.
- Exploratory programming that could grow into production code. This is actually an area where Lisp's nature really shines, but Clojure relies on the Java stack, exposing a lot of production code to it.
- Android development? Android development GUI development model relies heavily on class inheritance (meaning that you don't actually use it as a plug-in library; it forces you to follow a certain structure). It can be done, but it certainly isn't as natural as direct Java inheritance.

Some classical Clojure code:

```
(defn divides? [k n] (= (rem n k)
0))
(defn prime? [n]
  (if (< n 2)
    false
    (empty? (filter #(divides? % n)
(take-while #(<= (* % %) n) (range
2 n))))))
```

And a simple queue definition in the lisp way.

```
(defn make-queue []
  (atom []))

(defn enqueue [q x]
  (swap! q conj x))

(defn dequeue [q]
  (if (seq @q)
    (let [x (first @q)]
      (swap! q subvec 1)
      x)
    (throw (IllegalStateException.
"Can't pop an empty queue."))))

(defn queue-empty? [q]
  (empty? @q))
```

D

I used to love D. D is like C++ done right. D1 felt like a low-level-oriented Python. Like a pythonized C or something like that. It's awesome: you feel development speed, focusing on the algorithms and not the language, but you don't sacrifice low-level control when you need it. D2 brought a lot more of the complexity of C++, with the innovative touch of Andrei Alexandrescu. That made part of the community unhappy, albeit D2's focus on concurrency. D2 is not a clean language any more, but feels more like an experimental language with lots of untested features. I like it though, but I think its features pale in comparison with C++'s pervasiveness (once you have a more complex language). And also I think Go took the place that was once D's destiny. Walter and Andrei can't compete with Google, even if they can move faster and implement really cool things in the language. You can like D (as I sort of do), but I don't see a bright future for it. Just stick with C++ or go to Go for better native concurrency support. So, when would I use D?

- For developing a project from scratch, being able to interface C and to some extent, C++. You have to think in advance what those interfaces are going to be like, though. For instance, I wouldn't recommend it if you need to use a C++ GUI library, because that normally means dealing with C++ inheritance from within and that will throw all the advantages away. Just do this if you need C++ for a plug-in library (creating objects and using its functions, but no templating or C++ inheritance).
- If you need low-level programming with fast binaries. Again, doing your own thing, like a standalone program.
- If you want better native support for concurrency in the language.

Let's see some idiomatic D2, with pure functions, and immutable declarations.

```
uint grayEncode(in uint n) pure
nothrow {
    return n ^ (n >> 1);
}

uint grayDecode(uint n) pure noth-
row {
    auto p = n;
    while (n >>= 1)
        p ^= n;
    return p;
}

void main() {
    import std.stdio;

    " N     N2      enc     dec2
dec".writeln;
    foreach (immutable n; 0 ..
32) {
        immutable g =
n.grayEncode;
```

```
        immutable d =
g.grayDecode;
        writefln("%2d: %5b => %5b
=> %5b: %2d", n, n, g, d, d);
        assert(d == n);
    }
}
```

The max element of a list.

```
[9, 4, 3, 8, 5].reduce!max.
writeln;
```

It is definitely more expressive and a cleaner language than C++, by far.

Erlang

This is a very specific-purpose language. Erlang's web page describes it very clearly: […] build massively scalable soft real-time systems with requirements on high availability. Some of its uses are in telecoms, banking, e-commerce, computer telephony and instant messaging. Erlang's runtime system has built-in support for concurrency, distribution and fault tolerance. Erlang has been proven for, and it's behind some very demanding applications such as WhatsApp. The code itself feels very functional, and its syntax is clean and very readable.

Take a look at the code for a simple concurrent program:

```
-module(hw).
-export([start/0]).

start() ->
   [ spawn(fun() ->  say(self(),
X) end) || X <- ['Enjoy',
'Rosetta', 'Code'] ],
   wait(2),
   ok.

say(Pid,Str) ->
   io:fwrite("~s~n",[Str]),
   Pid ! done.

wait(N) ->
   receive
       done -> case N of
           0 -> 0;
           _N -> wait(N-1)
       end
   end.
```

Go

I haven't used this personally. Yet. But it's clear that this is Google's take on making a C with the good parts of C++ and better than both in its concurrency support. It has better features than C++, and it is way simpler. It has no unsafe pointer arithmetic, closures and first-class functions, and garbage collection. Go might become the server language in the future. So, when would I try Go?

- For server applications that need very high reliability and performance. This includes web apps.
- For highly-concurrent code that requires low-level control (otherwise, I'd stick to Erlang).

Go concurrent code:

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func main() {
    words := []string{"Enjoy",
"Rosetta", "Code"}
    rand.Seed(time.Now().
UnixNano())
    q := make(chan string)
    for _, w := range words {
        go func(w string) {
            time.Sleep(time.
Duration(rand.Int63n(1e9)))
            q <- w
        }(w)
    }
    for i := 0; i < len(words);
i++ {
        fmt.Println(<-q)
    }
}
```

Haskell

This language truly feels as a more advanced thinking tool than the others in this list. It has libraries for almost any need and it has a hard-core community. Arguably it's a language with a high barrier of entry. It will expand your mind, and surround you with some of the brightest minds in the programming languages communities, in my opinion.

I think Haskell is well worth learning, even if you don't build any real program with it. Being a relatively obscure language, I chose to classify it as reasonable since it is actually used in several areas, and especially in the financial industry. Haskell's code tends to be very compact and expressive, albeit a bit abstract, in the sense that you need lots of functions that are actually conceptual operations rather than steps of a process. I personally don't like its syntax (I think it has way too much syntax), but at least it serves a purpose and doesn't feel like clutter (I'm looking at you, Perl!). This language feels beautiful and coherent. Take a look by yourself:

```
binarySearch :: Integral a => (a
-> Ordering) -> (a, a) -> Maybe a
binarySearch p (low,high)
  | high < low = Nothing
  | otherwise =
      let mid = (low + high) `div`
2 in
      case p mid of
        LT -> binarySearch p (low,
mid-1)
        GT -> binarySearch p
(mid+1, high)
        EQ -> Just mid
```

Java

The same as C#, but for the Java Virtual Machine. It was there first (in fact C# copied it), and it's sort of "the standard" object-oriented language in the industry. It's used for everything, from web apps to games. Almost everything except embedded device software, and perhaps high performance parallel computation software. It serves as the foundation for many other languages (specifically its virtual machine). Take a look at Processing for an interesting project, where a wrapper language (just sugar-coated Java) is used for education and digital art. When would I personally recommend you use Java?

- Mostly when you want to access a very large pool of developers and knowledge base, i.e., you want the software to be maintained by someone else.
- When you need a multiplatform virtual machine present in as many devices as possible.

JavaScript

The lingua franca of the 2010s, the language of the web. The funny thing is that while it was previously seen as a very defective and limited language, a more recent wave of programmers have shown the world that, following a set of good practices and using various techniques, it actually turns out to be a great language. Especially if you take into account all the libraries and implementations that make up for JavaScript's design mistakes or missing features (such as a module system). Thanks to this, we even have JavaScript for the server, which brought this beautiful symmetry backend/frontend to life, finally.

There is a lot of research and effort put into improving JavaScript performance and derivative languages that compile to JavaScript. This actually proves that community is one of the greatest (if not the greatest) assets a language could have. The funny thing is, you see myriads of libraries doing the same thing over and over, making it one of the most competitive arenas for a library developer. See examples as Grunt vs. Gulp, or the battalions of competing JavaScript derivatives (Coffeescript, Typescript, Livescript…). It's crazy out there.

OCaml

It's sort of like Haskell, but it feels like it's more willing to bend to the programmer's desires. When there is need, some compromises to its purity are made in benefit of easier solutions, for instance when the procedural/object-oriented approach seems to work best. There are companies using it, I guess just for this benefit over Haskell. Take a look at this little snippet:

```
let n_arrays_iter ~f = function
  | [] -> ()
  | x::xs as al ->
      let len = Array.length x in
      let b = List.for_all (fun a
-> Array.length a = len) xs in
      if not b then invalid_arg
"n_arrays_iter: arrays of differ-
ent length";
      for i = 0 to pred len do
        let ai = List.map (fun a
-> a.(i)) al in
        f ai
      done
```

Looks almost like Haskell, right? But then you have the imperative flavor in that for loop…

PHP

Don't just assume PHP is horrible. Be a good Spartan and inflict yourself the joy of PHP. The good thing is: if you enjoy programming in PHP, then you are a true programmer. And it's the language of the cheap freelance work.

When would I use PHP?

- If you want to have the largest pool of web developers available.
- That's it, no other reason.

Python

A pretty language. I definitely like its whitespace-based block structure: you don't need ugly semicolons all the time. I like this so much that I tend to write my JavaScript this way. But this is very much a matter of taste, and as a matter of fact is the very reason many people don't like the language. It's a clean language that tries to take the burden of the syntax off of your shoulders. While it is debatable that it succeeds at this, the language is definitely supported by a great community, which put it in a very strong position when compared to its pal Ruby. It's always hard to choose between these two languages, although Python seems more widespread, and a more sensible choice for a variety of fields and applications. When would I use Python?

- Web development.
- Scientific computing and data analysis.
- System administration and tools.
- Game/3d application scripting.
- Cross-platform support.

Ruby

Ruby on Rails. The single reason this language could ever be part of this list. Of course, nowadays it's easy to see it in many other projects, but it all began with Rails. Before that, Ruby was an obscure programming language from Japan. This is a perfect example of how a killer app/framework spawned a great community which in turn made more killer apps/frameworks and made the language popular even though the place for this sort of language was supposedly taken.

One thing I've heard from many Ruby developers and I had experienced myself, is the actual joy that comes from using it. In other words, it's the contrary of a frustrating language, although I don't know if this is something from the language or Rails itself. The guys at metasploit seemed to have it very clear since the beginning as well.

Scala

Seems to be winning the race for the best JVM-based language award. I'm pretty sure that most of it comes from a familiar syntax, when compared with Clojure, the other big contender. As in Clojure, the reason this language is in this list is because its easy interfacing with Java make it a viable choice for a real project. Look at this small snippet generating the Hofstadter Q sequence:

```
object HofstadterQseq extends App
{
  val Q: Int => Int = n => {
    if (n <= 2) 1
    else Q(n-Q(n-1))+Q(n-Q(n-2))
  }
  (1 to 10).map(i=>(i,Q(i))).
foreach(t=>println("Q("+t._1+") =
"+t._2))
  println("Q("+1000+") =
"+Q(1000))
}
```

Scheme

This is probably a controversial language to be on this list, but I have an explanation. The three main issues associated with this language are:

1. Lack of one true implementation and multiple competing ones of dubious quality.
2. Lack of libraries.
3. Poor performance.

Well, the first one is partially true (there are too many implementations), but there are only a handful of good ones and you need to choose the one that best fits you. The second is also partially true: there are libraries, but they are scattered. There are some projects that offer alternatives, and lots of tiny projects out there. The fragmentation of the language is made obvious when looking for support code: you need to make it work

with your implementation. However, this is often not so difficult or time-consuming, and most importantly, if you use Scheme implementations with good FFI support, such as Gambit or Chicken Scheme, you have easy access to all those libraries in C. I actually do it, and it works great, contrary to what you may think. Finally, poor performance. This one is actually completely false. Implementations such as Gambit are very fast, and you have plenty of options for optimization (starting from algorithmic optimization, global Scheme compiler declarations, and of course, C code that can be easily interwoven with the Scheme code when necessary).

Yes, I'm fanatical about Scheme, however, I admit it has one deadly weakness: it's an awful language for sharing code, with a not-so-good community. It's so flexible that every single programmer wants its own little perfect solution to the task. It's the complete opposite to Java: great for individual/small-team projects, prototyping and exploratory programming, unproven for large teams. But in those situations, is perfectly fine, and you can actually ride an extremely fast and pleasant development cycle. Lastly, another very interesting feature of the language: you can easily compile to JavaScript with one of the Scheme-to-Js compilers, so you could enjoy the same kind of symmetry that you get when developing with Node.js on the server. The following are concrete examples where I'd use this language:

- For exploratory programming, when I don't exactly know where I'm heading.
- For fast prototyping of ideas that don't really require a large library only available in languages like Python or Ruby.
- For scripting large programs or platforms developed in C/C++.
- For building an application with large portions that need to be written from scratch.
- For games and OpenGL/ES-based multiplatform applications.

Here you have three examples of Scheme code. You actually have to implement these functions yourself as they are not directly available across all implementations, even though they are rather useful and common enough. These would work across all implementations (provided they are R5RS-compatible).

```
;;! Recursive map that applies
function to each node
(define (map** f l)
  (cond
   ((null? l) '())
   ((not (pair? l)) (f l))
   (else
    (cons (f (map** f (car l))) (f
(map** f (cdr l)))))))
```

```
;;! Explicit currying of an arbi-
trary function
(define (curry fun arg1 . args)
  (if (pair? args)
      (let ((all-args (cons arg1
args)))
        (lambda x
          (apply fun (append all-
args x))))
      (lambda x
        (apply fun (cons arg1
x)))))

;;! Implementation of filter, with
the match macro
(define (filter p lst)
  (match lst
   ('() '())
   (((? p) . tl) (cons (car lst)
(filter/match p tl)))
   ((hd . tl) (filter/match p
tl))))
```

If you solve the issue of a minimal development framework (yourself or via projects like Scheme Spheres, [schemespheres.org] then you are on a flywheel.

Conclusion

I started this long post with my selection of choice. Programming is a beautiful craft that I love whole-heartedly, so I admit that I'm heavily biased according to my personal experience. Choosing a programming language for a task or project is sometimes difficult, as so many variables take place. In my opinion, there are three that prevail, in this order:

1. Is the project aimed at production, or belongs to a sufficiently large organization with a culture or bias towards a programming language?
2. Is the task at hand sufficiently special to require a programming language with very specific features?
3. Do you love or want to try developing in that programming language?

That's how I approach this issue. Even though sometimes I break the rules… ■

Álvaro Castro-Castilla is an architect-civil engineer, digital artist and software developer. In his trajectory he has been playing with and within the frontiers of these disciplines, exploring and creating worlds that can be built with code.

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

I NEVER FINISHED ANYTH

By JAMES GREIG

I'LL OFTEN COME up with an idea that I get excited about.

Then I brainstorm a catchy name for it, check the availability of urls and social media accounts, maybe even set up a landing page. It gives me a big rush, and I imagine a dazzlingly bright future ahead for the concept.

And then the idea crawls up and dies inside of me.

Why?

Because I don't actually do anything.

To finish things, you need to fall in love with the part of the process that's harder to love — the bit where you roll up your sleeves and do the damn thing.

Maybe that's why it's got another much tougher sounding name: execution.

The human brain is a brilliant idea-generating machine. In the past we had to convert our ideas into solutions just to stay alive: to make sure that we had enough food... or didn't get eaten. But now, in the safety of our comfortable, hygienic, homogenized 21st century lives, it's all too easy to fall asleep on our true potential.

Wake Up and Smell the Hard Work

Your idea doesn't mean diddly-squat until it's out in the world. And to do that is going to take some hard manual labor.

So to stay on track, you'll need to engage with the execution process as much as the idea itself.

None of my various bright ideas — a social network for sneaker collectors, customizable artwork of your bicycle, a recipe sharing platform, a book about designers turned entrepreneur (OK, that last one I am actually set on doing) — have come to fruition yet.

And whilst CycleLove (and its sister shop CycleLux) might be building momentum, I still have a huge hang-up about creating the eBooks or information-based content about cycling or whatever it is that I've been talking

about for months and months. It's still a blog, not a business, and costing me money instead of making it.

I chickened out of the work.

You need graft, or grit, or gumption, or whatever you want to call it.

Whether it's by actually blogging on your blog, or starting your startup, value is created by doing.

> *It's easier to sit around and talk about building a startup than it is to actually start a startup. And it's fun to talk about. But over time, the difference between fun and fulfilling becomes clear. Doing things is really hard — it's why, for example, you can generally tell people what you're working on without NDAs, and most patents never matter. The value, and the difficulty, comes from execution*
> *— Sam Altman*

Dial Down the Resolution(s)

When I looked back at the list of goals I'd set out for 2013 the other day, I felt pretty embarrassed. Especially as it's published in plain sight on the internet. I didn't come close to achieving any of my resolutions. Not one thing on the list.

But I know that beating yourself up about this kind of stuff is stupid. (Make changes, not criticisms).

So…I haven't made any New Year's resolutions this year.

You don't want high resolutions anyhow — you want low resolution.

You want to let go of the fear of fucking up, of it not being perfect, of what other people think, of things that probably won't ever happen, and just crank that stuff out, baby.

Instead of Trying to Finish Everything, Try to Finish One Thing.

Today if possible.
And then another...
And another...
And...
(I think I just finished this article).

What are you going to finish today?

■

James Greig is a London-based graphic designer/writer [greig.cc] and the founder of CycleLove [cyclelove.net]

Reprinted with permission of the original author.
First appeared in *hn.my/anyth* (greig.cc)

BETTER SENDING, BETTER INSIGHTS

ANALYZE, REACT, ENGAGE

NEW

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

mailjet™