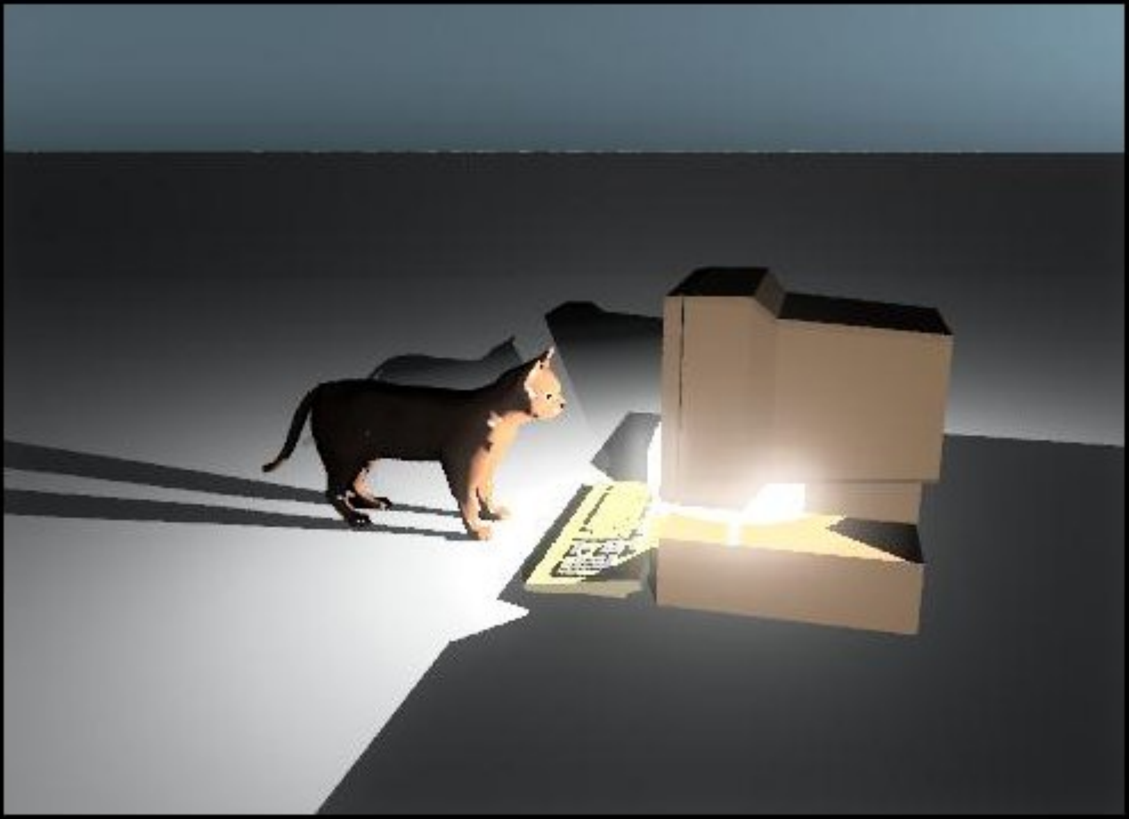
Technicat on Software



Phil Chu

Technicat on Software

Essays on Software Development and other Good Stuff

by Phil Chu

Introduction

This is a collection of essays various aspects of software development that I wrote on my software consultancy web site over the past eight years, partly to kill time between projects, partly to practice writing, partly so if someone wanted to know my opinions, I could just point them to the essays, and largely as a form of therapy after two decades in the business. To my pleasant surprise, I've received some complimentary emails, occasional spikes in web site traffic due to getting digg'd and reddit'd, and even found some of these essays used as college course material!

Apologies to Joel on Software for my lack of imagination in the title. I confess Joel Spolsky's collected writings are an inspiration for my attempts. While I'm at it, I also recommend reading the books by Fred Brooks, Gordon Bell, Henry Petroski, Richard Rouse, Richard Gabriel, Steve McConnell, Lawrence Lessig, Paul Graham, Eric Sink, and Scott McCloud, and the Nolo books. Also, read **Dilbert** and watch **Office Space**.

For more on these writings, visit http://technicat.com/.

The cover image of this book was created with the text-to-3D generator on http://wordseye.com/ . Try it out!

Seven Habits of Highly Effective Programmers

As a software engineer, you might want any number of things out of your job - a steady paycheck, the opportunity to work on interesting projects, a springboard to the next better job, or maybe you just like hanging out with other programmers. But by "effective", I mean the ability to complete projects in a timely manner with the expected quality. After working on dozens of software releases, I believe the following practices will bring you there, and while they may involve sticking your neck out, I'd like to think they will also advance your professional reputation, career longevity, and personal satisfaction.

Understand Your Requirements

The first step in becoming an effective programmer is to ensure that you are spending your time wisely. And there is no greater waste of time than in working on something that is not useful or never shipped.

Build Early

Get a demonstrable system working as early as possible. This means establishing the interface first, whether it's an API or user interface, and stubbing the encapsulated functionality as necessary. This allows your "customers" to check it out, by exercising the user interface or writing code to the API, and any inconsistencies or omissions in the initial spec can be detected early. Chances are, you will notice problems or potential improvements even before releasing this first deliverable.

There is a classical school of thought that believes if you design everything up front, then all you have to do is write the code and you're done. That works great if you've done the exact same project before. Otherwise, it's more likely you'll run into a point where you're just guessing or operating on questionable assumptions.

> Upon joining an early-stage wireless internet startup, I found myself in two months of design meetings for a wireless portal and gateway due to launch in six months. Eventually we got tired of meeting and finally started coding. Within two weeks, my part of the project had no resemblance to the original design, and the first wireless connection test two months later revealed a fundamental misunderstanding of the wireless protocol.

This is not to say that design is unnecessary. But after a certain point, design is just speculation. Design should be validated with implementation, and better to do that early and continuously than late and, well, too late.

Even if the original design is sufficient, once you have something you can tweak, you can improve upon it. Hardware products (who designed this VCR?), buildings, and large-scale software projects suffer from interfaces that were frozen in "preproduction", but with software, you have an opportunity early in the project to refine your understanding of the requirements and produce a suitable interface. But it must be done early.

Getting something ready early is also good for your occupational well-being. Your boss will appreciate seeing evidence that something is actually getting done and having something available to demo. On the other hand, a drawn out period with nothing to show is a recipe for anxious management.

Deliver Often

Once you have something working, don't just leave it as a "proof of concept". Let people play with it, see their reactions, and let this guide and prioritize your development. There is no substitute for watching how people use your software. Customer questionnaires and focus studies might provide some useful input but run the risk of transferring feature and design decisions from the developer to the customer.

In particular get the software into the hands of the QA staff as soon as possible and feed them regular builds, preferably at scheduled intervals. Having them test automated daily builds is ideal, but even a weekly build is pretty good. This will help them feel involved in the full life-cycle of the project and they should be best-trained at identifying and reporting problems. The highest priority should be given to issues that prevent them from using the product, e.g. crashes or deadend paths - you want them to cover as much as possible as soon as possible and get a feel for the whole product so design issues can be identified early.

> At a small 3D graphics software vendor, I was put in charge of porting the flagship product from SGI workstations to Windows NT. After six months, the port was so incomplete and crash-prone that I was reluctant to give the first "alpha" build our test group. Fortunately, the QA manager insisted, and the resulting bombardment of bug reports forced me to immediately focus on the problems that prevented the testers from exercising the application in any meaningful way. Left to my own devices, I would have worked on what seemed to be the harder and more important core 3D issues, and probably delayed too long on seemingly mundane issues like the user interface, load-save functionality, and compatibility with all the varieties of consumer hardware we were planning to support.

Programmers often don't want to release code to testers early - they don't want to hear about a bunch of bugs they already know about, and quite possibly the testers don't want to test something that barely works. But it's the testers' job to find these problems and programmers need to realize bug reports are a good thing, if they arrive early enough.

Keep It Real

Keep your software running in as close to a shipping state as possible. You never know when you'll have to demo the system, send out an evaluation copy, or even deliver ("OK, time to wrap things up!")

Use Real Data

If you just test with sample data, that big iceberg of real data out there is going to sink your program.

> One of the leading semiconductor fabs evaluated a supply chain management product I was working on. After crunching out a milestone delivery to them, we got word back that the first batch of data they fed it from their own operations was still processing - for two days. I sympathized with the lead programmer, who had to dig down and emergency-optimized everything he could for two weeks with both management and client breathing down his neck. I'm just glad it wasn't me on the line.

Use Real Builds

Remember the development build on your machine is not the real build.

> On a recent game development project where I worked on the user interface, I got intermittent reports from QA that some colors were not correct. Eventually, I realized the problem only showed up in release builds and another programmer used the special console debugging hardware to track down the bug. Which turned out to be a silly mistake I'd made two

> months previous, failing to specify an initial color value in a few cases. The debug build always selected a specific default value, while the release build optimized that away and the result was less determinate. If I'd made a point of running the release build frequently, I would have spotted my mistake immediately, instead of losing it in the sands of time.

Merge Often

Don't procrastinate on merging your code with the main code base - the longer you wait, the harder it gets.

> I worked with a programmer who "couldn't be bothered with" all the new code and data changes that showed up in the repository every day. And certainly, daily merges did take up some time for all the other programmers, and this programmer was able to run some impressive standalone demos with a snapshot of the code and data. But every time we had a milestone delivery, it took days to get the isolated code reattached to the current codebase again, sometimes compromising the milestone delivery and risking the funding for the entire project.

Keeping your code out of the official build means that programmers cannot evaluate your code and testers cannot spot bugs early. Maybe you don't want people picking on your code or bugs, but it's better to identify those issues early than later - suck it up.

Understand Your Code

Life is full of wonderful mysteries, but your code is not the place for them. You don't have to know how your car works - if the engine starts making strange noises, you drop it off the mechanic. When it comes to your code, if you don't understand how it works, or doesn't work, no one will.

Code with Style

My childhood piano teacher once commented to me, "Your sister has a good sense of timing, and your brother has a good feel of the keyboard." Then he paused. "You, uh, you work hard."

Programming is one of those things that a lot of people are more or less competent at, but some in particular have a flair for it. I'm a lousy piano player despite years of lessons, and I'm a mediocre basketball player although I enjoy playing it immensely. But I do like to think I have a flair for programming and writing. And not surprisingly, I think good programming is like good writing. Both prose and code are textual, have grammar, syntax, spelling and semantics and spelling. For most coders and writers, this is enough, but the best writers and coders have an esthetic and their work features structure and style that can often be identified with the author.

> Many Windows programmers wonder why grumpy old Unix/Mac/Amiga/Lisp programmers rail against Win32/MFC/.NET, but if all the API's you've seen are from Microsoft, you probably don't know there's anything better.

Perhaps not everyone is capable of writing stylish code - I've heard it said that good object-oriented programmers, in particular, are born and not made. But like fine music, wine, and literature, you can learn to appreciate good code.

Cut-and-Paste

The opposite of stylish programming is cut-and-paste. Grab some code from somewhere that is supposed to do something like what you want, tweak it until it sort of works, stir, repeat, and voila, you have the software equivalent of mystery meat.

> A few months after leaving one company, a former coworker emailed me a single function consisting of ten pages of cut-and-paste code and asked why it wasn't working. I could have very well asked why it should work at all. If you can't explain how your own code is supposed to work, how can you expect anyone to help you with it? (He has since moved on to a management position at Microsoft)

I've even had trouble diagnosing my own code that was cut-and-pasted from sample code. It's a reasonable way to start new code, but you can't just leave it alone when it seems to work - you have to go back and make sure you understand it line by line and clean it up for your own purposes.

Keep it Clean

The key to keeping your house/condo/apartment clean is to spend a little time cleaning it every day, or at least every week. If you wait until your abode is an unsightly mess, it's just too much damn trouble to clean it all up and you end up just doing a halfhearted job. Or your hire a cleaning service.

Assuming you don't have the luxury of hiring someone to come in and clean up your code every week, you should periodically inspect your code, sweep up accumulated hard-coded numbers, outdated comments, misleading function names, or you'll inevitably end up with uninhabitable code that's embarrassing to show anyone else. And if you're not embarrassed, well, you should be.

> One programmer I supervised kept reporting to me that her code was "done". This is what management normally likes to hear, but it drives me crazy. Code is never done - you have to debug it, maintain it, evolve it until it's put out to pasture.

Questions? Comments?

Some like to think of programming as a craft. Others, engineering. More often than not, it's archaeology. You dig through sediments of code and wonder what purpose all these strange artifacts served. Do future generations a favor and leave some clues.

> I asked the aforementioned engineer whose code was "done" to add comments. The result, a function named GetData was prefaced by the comment "Gets data." That's not just useless - it's insulting. What data?

> (factory automation schedules) In what format? (a proprietary XML format) From where? (an in-house server, using TCP/IP) Not to mention little details like what happens when the server is unavailable or the transmission is broken.

Document your code as if someone else might have to take it over at any moment and know what to do with it. That person might actually be you - how often have you had to revisit your own code and thought to yourself, what was I trying to do here?

> On a contract with a previous employer, I was asked to look over a piece of code that no one else had time to attend to. At first, I thought it was a mess and didn't know what was going on in there. Then I gradually figured out what the code was doing, and I grudgingly admitted the code wasn't too bad. And then I eventually realized that I had written the code two years ago. Note to self: need more comments.

With that in mind, annotate your code as you write it, instead of waiting for a convenient cleanup phase in "post" - annotating as you code can even clarify your thoughts while you're programming. You can be your own pair-programming buddy.

As a bonus, these days you can generate nice HTML or otherwise-formatted documentation from source code comments, using javadoc, doxygen, whatever. Ideally, the doc-generation is part of your nightly build and available via your intranet.

Full Warning

Ignore compiler and runtime warnings at your own peril. They are called "warnings" for a reason.

> I shipped one Unix-based application that had a problem linking some functions successfully - we worked around it by relinking those functions at runtime. When we performed a clean rebuild six months later for the next release, it was revealed that we had turned off linker warnings which would have alerted us of a known linker bug. In our defense, we had swept the linker problems under the carpet at the Unix vendor's suggestion, (thanks, SGI!) but it turned out we could get the link to work just by reordering our libraries.

Crank up the warning levels on your compilers, sprinkle your code with assertions, and log the build and test-time warnings. Better yet, include a count of those warnings in your build metrics so you know if you're dealing with them or letting them accumulate.

Optimal Programming

Code with Purpose

On the other extreme from cut-and-pasters are those who change code just to make it look prettier (at least to them). While it's laudable to have a programming esthetic, it's a waste of time (and a useless risk) to change code just so it looks better to you. It's aggressively annoying to go through and change code that other people have written just so it looks better to you.

> A fastidious coworker of mine went through our codebase and removed all the expletives. Probably no one would have complained if he had just cleaned up the code written by the entry level employees, but the expletives belonged to the technical lead of our group who was also one of the few distinguished Fellows at the company.

Do No Harm

"Refactoring" is all the rage, now, but programmers often take it to mean any code cleanup or redesign. The trick is in reorganizing code for the better without breaking anything. If you break existing functionality in the name of progress, you're sending one of two messages: 1) your time is more important than everyone else's, or 2) you're incapable of touching code without breaking it.

> I had one particularly contentious coworker who decided to reimplement the parser in our system but left the code in an unbuildable state by everyone else. I asked him to revert his changes and then found the code was buildable but not runnable - asked about it, he replied that he removed the parser entirely "per your request". Not a team player.

Keeping the code functioning takes some patience and extra work - you have to be diligent about regression-testing your work and chances are you'll need to keep old code and interfaces around for a while as you migrate functionality to your new code. But for everyone to work with the same codebase, that's what you have to do.

Find the Bottleneck

People always talk about "optimization", but that isn't really a correct word. We're rarely shooting for the optimum - instead, we make improvements and tradeoffs to achieve good-enough performance.

> I was asked in a phone interview with Google how I would search for a number in an array of ordered numbers. Obviously, the questioner was asking for a CS 101 answer - binary search. But in real life, I would probably do the "wrong" thing - search the array from beginning to end. There's no point in taking twice the time to write twice as much code that has to be maintained and debugged if the application performance is good enough, and in particular if that piece of code is not the bottleneck in the application. (And I seriously doubt you'd have that data laid out linearly in a fixed array like that if it was the bottleneck)

If you do need to optimize for speed or space in you application, attacking anything other than the bottleneck is a waste of time.

Manage Thyself

You probably have a lot of complaints about your boss being a lousy manager, and you're probably right. So you have to be your own manager. Even if you have a decent boss, he's not going to stand behind you telling you what to type and how fast (although I'm sure many would like to).

Are We There, Yet?

Programmers are notoriously inadequate at providing useful schedule estimates. I think this is a bad rap, since management, left to their own devices, often make even worse predictions, and unwelcome news from engineers is often ignored. (A common theme in any engineering disaster). But still, awareness of the schedule is critical to actually getting the project done on time.

> On one commercial software project, some of my coworkers were blissfully unaware of the product release date - one inquired whether it had been released already, another was surprised to find it was going out in a few days.

The worst, and most common, schedule estimate that a programmer can give is "it should just take a couple of days". Every time I hear that, even from my own mouth, I wince.

> The president of a graphics software company really wanted support for VRML (at the time it was the Next Big Thing) included in the product we were releasing in two months. He probably figured (correctly) that I would resist starting a new feature, so he went to another engineer and got the answer he wanted: "a couple of days". Two days later, I told the president we-just-wasted-two-days-of-his-time-and-mine-while-there-are-two-hundred-high-priority-bugs-to-fix, which he found to be a persuasive argument. (postscript: VRML didn't exactly take off like gangbusters)

And then there are programmers who are unable to come up with time estimates at all. But there's no need to get thrown off by the fuzzy nature of the request - it is an estimate after all, and in fact you should avoid using formulas. If you're an experienced engineer, you know how long comparable tasks have taken you before, and if you're not experienced, you can ask someone who is.

> A smart friend of mine who was often assigned to developing experimental prototypes asked me, "how can you schedule research". I think it was a rhetorical question, but even pure researchers have schedules. Someone is paying them and expects results, even if it's a number of demos or published papers in a certain timespan. And if you really don't have the foggiest idea how long something will take, then you're the wrong person for the task.

Sometimes programmers are reluctant to commit to a schedule because they're afraid of the accountability. It is true, in this imperfect world, managers will try to bargain you down on schedules, political factions may saddle you with tough or unrealistic schedules in the hopes that you will fail, and it is a sadly common story that after you commit to a schedule, you don't get everything you need.

> I had one boss who after asking for an estimated completion time would say, "do you promise?" But ask for a commitment on the required hardware and other dependencies, and it was "I'll try."

All I can say is, stick to your guns and give a realistic prediction. Any concessions should be based on pragmatic tradeoffs between features and resources. Be clear about the assumptions, dependencies and resources on which the schedule is based, and get it written down somewhere so you can jog defective memories later.

Plan Your Progress

You wouldn't just hop into your car before deciding where you want to go, right? And probably you have a route in mind before you start driving, too. Similarly, before you sit down at our computer, you should know what you want to accomplish that day and have some idea how.

Distractions will come up day-to-day, so you won't always be able to accomplish what you want. And contrary to those who treat software engineering groups as vending machines (they would probably shake us vigorously if they could) some tasks take more than a day. So think about what you want to accomplish by Friday, and if you do, then you can enjoy the weekend that much more.

Continuous Education

A corporate soccer team member once asked me, as we were lacing up our cleats, "what's the secret to C programming?" If there was such a secret, I'd be hawking it on late night TV along with ab machines and how to get rich in real estate. Sorry, there's no shortcut - you have to learn and practice and make some mistakes. And you don't necessarily have to rely on corporate training or going back to school - there are plenty of (inter)national and local professional groups, books, and of course, the Internet.

It's Science

It's called "computer science" for a reason. It's easy (maybe too easy) for anyone to to start programming, without a formal computer science education. In particular, those from other engineering and science disciplines can pick up programming quickly and make a good living. But to effectively tackle non-trivial tasks, you need to know the inherent capabilities and limitations of software and recognize prior work, so you don't waste time reinventing the wheel, badly. You don't have to know everything under the sun, but you should have at least a cursory familiarity with many areas and be prepared to do some additional research as necessary.

For example, anyone who creates a new file format should know something about compilers. I don't mean all the code-generation optimizations like loop unrolling, but the basic issues and various phases of compilation and most of all the importance of specifying the tokenization and grammar. Nowadays, most people by default will use XML, and that's a good thing, but before then it was typical to cobble up some text format, point to some generated sample files as documentation, and then everyone else who wrote another parser would cobble something up that would read in some files but not all. In the problematic cases then you could point fingers either way - either the reader is bad or the writer is bad. Whichever product is more popular wins.

> One of my pet peeves with the 3D graphics industry is the plethora of ill-defined file formats. When I implemented an OBJ file parser for a 3D content creation product, each exporting product that I tested against generated markedly different files, using different whitespace and newline conventions, for example. In refreshing contrast, a coworker of mine fresh out of school designed a new game interchange format using a grammar and lexer specification. (These days, it's not much of an issue anymore - most new graphics file formats seem to be based on XML.)

And if anything differentiates programmers who can just put together simple scripts and user interfaces and those who can tackle real problems, it's an understanding of computational complexity, i.e. how algorithms scale with the size of the problem. Every programmer should know basic complexity terminology and have a general knowledge of the complexity of common problems.

> My first job was in computer-aided semiconductor design, which has a lot scalability issues, including some NP-complete (essentially intractable) problems. But some of the engineers would run around excitedly saying "it's the traveling salesman problem!" every time they saw a problem that couldn't be solved in linear time, and in other cases we boasted of "linear-time" algorithms which probably meant linear-time most-of-the-time. Or some of the time.

Free Beer, Free Speech, Free Software

OK, there's no free beer, but this is a good time to be a programmer (well, recession and outsourcing controversy notwithstanding) - just about everything you need is on the Internet tutorials, discussion lists, and free software. All you need is the hardware and a broadband connection.

R-E-S-P-E-C-T

One requirement for being an effective software engineer is to be taken seriously. You need to have the respect of your peers and managers, at least for your technical capabilities, to have control over your own work and influence over others.

There is Such a Thing as a Stupid Question

Really, there are lots of stupid questions. Asking intelligent questions that enhance others' respect for you is a professional skill. A good question that exposes unresolved issues tells people that you know your stuff and you're sharp enough to catch all the implications. Asking for clarification about a specification shows you know how to find and read the spec and your ability to detect ambiguities.

If you don't get any answers to your question, chances are there's something wrong with the question, so don't just repeat it. Ask the question differently, with more specifics, or more background. If you've been on the other end of a technical support line or even just spent time on discussion lists answering newbie questions, you'll appreciate the consideration.

> I pride myself on cultivating good relations with developer support staff by submitting elaborate bug reports and precise questions. But I do remember one lapse where I tossed out something along the lines of "What's the deal with that issue that came up several weeks ago?" You can imagine the prickly response - "What do you mean by what's the deal, and what issue are you talking about?"

It doesn't pay to be rude, especially if you're essentially asking for free tutoring or consulting on a discussion list. Even if you're asking under the auspices of a support contract, irritating your support contact isn't going to help you in the long term.

> I used to take pains to explain to belligerent newbies why their questions didn't make sense or what they were fundamentally doing wrong. Now, the bozo filter kicks in quickly - one "All I want to know is....", and they're ignored.

Let everyone know that you read the documentation and googled the subject. Besides avoiding the inevitable "RTFM" and "Google is your friend" responses, this shows you've done your homework and those who want to be of assistance don't have to search through the same resources. If you do expect them to search through those resources for you, then you're saying your time is more important than theirs, and you are just one more perpetrator of the "tragedy of the commons".

There is Such a Thing as a Stupid Answer

If you're going to act like you know what you're talking about, you really better know what you're talking about. Engineers sometimes communicate more to show off their own knowledge rather than to inform (although, if you can do both, kudos to you). This is often inflicted in employment interviews, under the guise of "finding out how you think" the candidate is asked inane puzzle questions. This can backfire, though, if the candidate has any self respect.

> One CTO interviewed me over the phone by asking me to sketch out the resulting stack frame from a C++ compilation and then report the result back to him verbally. We went through it step by step and every time I gave him a correct answer he asked me to give a wrong answer instead so we could go over why that choice wouldn't work. I couldn't tell if we were trying to demonstrate how smart I was or how smart he was.

There's also the blame game. As an engineer, you can't rely on your money and looks - all you've got is your credibility. So if you make a mistake - 'fess up.

> I had the privilege of working with a senior engineer who was never wrong. When his Java code was crashing on multiprocessor systems, it was Sun's bug. When I took over the code and pointed out the UI code was not supposed to run in multiple threads, he insisted there was only one thread. When I listed the seven threads (that I could find) in the code, he agreed I shouldn't have all those threads and I'd better fix it. He programmed in that fashion too - he didn't fix any bugs, he just covered them up with more code.

Finally, a bit of time-saving advice: Don't get dragged into stupid arguments. Stupidity is contagious.

Management Means Never Having to Say You're Sorry

> "I have people skills; I am good at dealing with people. Can't you understand that? What the hell is wrong with you people?" - Tom Smykowski, in *Office Space*

Know Your Stuff

The saying goes, those who can't do, teach. The same could be said of management. (And those who can't manage, consult? But let's leave that for another essay...) In any case, I've often complained long and bitterly about software project management, and a coworker once responded, "Phil, you're always right". I don't know if he was serious, but in the spirit of that encouraging or sarcastic endorsement, here are my recommended practices for managing software projects.

In many cases, "project manager" is a Microsoft Project manager whose apparent sole responsibility is to maintain a Gantt chart. (Sometimes even that is too advanced - I've sat in meetings watching people fumble with an Excel spreadsheet) But there's got to be more to project management, or any idiot could do it.

Know the Technology

The best way to gain the respect of your engineering staff is to understand the technology they're using. This isn't the same as thinking you understand. A little knowledge is a dangerous thing, especially the latest knowledge gleaned from buzzword-laden articles in business magazines.

> At an early stage on one large application running in Lisp, I mentioned to my boss that I thought garbage collection was causing some performance problems. From then on, every time he saw a slowdown, he'd ponder, "hmm, looks like garbage collection."

As a manager, it's unlikely that you know more about everything than everyone on your team, even if you're the chief high tech mucky muck of your organization. So take the opportunity to learn from your people.

> I have a friend who graduated with a PhD from a well-regarded computer science department and yet was treated to an impromptu lecture on computer simulation from his company's president, who had lesser engineering credentials from over a decade ago. Not surprisingly, my friend was rather offended.

Worse is when your lack of expertise causes you to miss problems in the project design and execution.

> As a technical lead for a subsidiary of PRIA, a supplier of semiconductor fabrication equipment, I correctly chose XML as a platform-neutral format for our fab scheduling system, but didn't learn enough about XML until after the project was delivered to realize that we didn't do a very good job of sticking to XML (e.g. no DTD). Too bad - this was back when XML was pretty new stuff, so if I'd really known what I was doing, maybe I could have defined an industry standard format for semiconductor fab automation.

Master Your Domain

Even if you don't know much about the engineering aspects of your project, you'll have value and get some respect if you know the application domain.

> As a development manager at a computer graphics software company, I visited one of our tool vendors and gave them an impromptu demo with the intent of getting them excited about supporting our product. Unfortunately, I couldn't figure out what more to do after creating a cube and making it spin. Should have rehearsed, I guess.

Knowledge of your application domain is important in designing a useful product, supporting your customers, and avoiding embarrassment.

Know Management

You should also be familiar with the state of the art in software project management.

> If there's one classic textbook on software engineering, it's Fred Brook's *The Mythical Man Month*, and if there's one classic software engineering principle anyone remembers from that book, it's the rule that adding more engineers to a late project just slows it down. I don't know if anyone I've

> ever worked for has read that book, but I have been on more than one project where panicked management piled on additional staff.

Of course, software engineering is just as trend-filled as other engineering fields (TQA, Six Sigma, ISO 9000...), and over-hyping is the rule, but you can't intelligently discuss practices and choose among them and apply them without reading up on them first. And if you don't know management stuff, what do you bring to the table - charisma? You're just an engineer who's not doing any engineering.

Good Management is Risk Management

The most crucial aspect of project management is risk management. Every decision made in a project should be weighed against how much risk it adds to the delay or failure of the project.

If I Had a Nickel....

I've often joked that every feature request should be accompanied by a dollar, but I betcha if managers had to whip out their own billfold and pull out a dollar for those "important" requests, the feature set would shrink dramatically.

> I was working on a submarine simulator that, up to that point,only had navigation controls. When our manager said he wanted to fire a torpedo for tomorrow's demo, we hastily improvised a reasonable facsimile. Encouraged by that, the manager innocently asked ten minutes before the demo whether he could fire a cruise missile. One of my coworkers grabbed him and pulled him out of the room before I could say anything.

There is no free lunch - everything has a cost, but it's usually felt by the poor slob who ends up working the weekend rather than the person in management or marketing who belatedly said "we really need this".

Scotty Doesn't Work Here

I blame Star Trek. Aliens are always humanoid and speak English, and the off-world females find Kirk irresistible - no one believes that's the stuff of reality. But the last fifteen minutes of every episode where Scotty or Geordi jury-rigs an engineering solution based on a never-tested hypothesis, and it always works just in time to avert disaster.

> I knew I was in a Star Trek moment on a military distributed interactive simulation project (basically a big multiplayer video game) when the project manager actually referred to me as "Geordi".
>
> Sitting in front of a mock submarine console, wearing submarine coveralls while the PR cameras rolled and Navy VIP's watched, I pushed the "start simulation" button that would connect us with the entire wide-area networked simulation and watched the system core dump as it got flooded with packets from the rest of the exercise. The project manager calmly informed everyone that we had "technical difficulties" and to "stand down", while I tried not to puke. I fixed the problem in the standard

> episode final fifteen minutes by deciding we didn't really care what anyone else was doing and turning off our system's data read from the outside world.
>
> This project had the all the makings of a disaster - I cooled my heels for two months waiting for a security clearance, the networking hardware was two months late (it was a seven month project with a hard deadline), the laboratory had no experience on this type of project, and one of the three engineers working on the project didn't do anything (but amazingly kept repeating "I'll have it done tomorrow" for months). And then I spent the last month of the project working while flu-ridden. Our participation in the exercise was hailed as a success (evident when all the senior managers who had kept a healthy distance away suddenly show up), but I don't think I could physically take any more successes like that.

In real life, the Starship Enterprise would blow up every time - the episode-saving idea would be faulty, or Scotty (or Geordi or whoever, depending on which series you watch) is out sick, the warp core isn't performing up to spec, Jean-Luc didn't get the memo, etc. Depending on programmer heroics to pull you through the crunch time, and even worse, building such a crunch time into the schedule, is a recipe for disaster. It's fun to remember the successes, but the result is usually failure.

Get That Fresh and Clean Feeling

Make sure you build the product regularly and from scratch. A common practice is to do this as an automated nightly build. If you don't do it, then you never know if you can really do it.

> A group developing software for use with the Hubble Space Telescope decided to rebuild the currently-deployed version a a reference for our research in designing the next version. But the previous build had been made years ago, and there had been enough VMS Fortran compiler changes that the old software didn't compile anymore. If we actually had to fix a bug in the old software anytime in the past few years, we would have been out of luck.

It's no exaggeration to say that the fate of your company could depend on the ability to make a clean build.

> One tool vendor I used to work with went Chapter 11 and at least one rescue acquisition failed because they couldn't demonstrate a successful clean build from the source code.

Having a regular automated build proves that you have a functioning build procedure set down in a script, not dependent on some sequence of magical incantations resident in one build engineer's head.

Let the Code Set

I've never seen a code freeze really frozen to my satisfaction - there's always temptation to keep adding "just one more thing" until the final deadline. But instituting a freeze is better than coding pell-mell until the end, a sure recipe for a broken release.

> Game development is particularly notorious for crunch times, which poses greater risk as the projects get more complex. I was on one project where the last level was scheduled to be completed just before the game was to go "gold". Given my own stand was that asset creation had to be completed a month before the the release date, management did it's best to reassure me "it'll be all right", and indeed we did just complete it on time. I found out later that the level export performed on the day of the disc burn was missing key optimizations, so the game probably would be running at least twice as fast if we hadn't cut it so close.

The point is not to completely stop development, it's to slow it down in a deliberate manner to mitigate risk. It's really more a code "gel" than a code freeze. The alpha milestone should mark the feature freeze point - any feature that is not functional yet should not go in. Once the beta milestone is reached, every potential bug fix should be evaluated according to its benefits and risks.

Code isn't the only thing that should be frozen near the end of a project. The development environment should be stable, tool, i.e. you should have a tool freeze. New compilers, libraries and runtime engines can introduce new bugs.

Know Your Demo

Risk management is not just important for releases and milestone deliveries. Even demos should not be taken for granted.

> One company president kept bringing in bigwigs from the parent corporation for impromptu demos of an in-development product, even after I asked him to at least give me a couple of hours notice so I could make sure have the code running. Eventually, he strolled in, tried to run the product, crashed it and helplessly tried to restart it with the VIP watching and me studiously ignoring the proceedings. I almost felt guilty.

Stay On Target

Emerson not withstanding, inconsistency is a dangerous thing when exhibited by management.

Unmix Your Messages

Corporate vision statements tend to be vague to the point of uselessness, but it is important to be clear about what's important and to be consistent about it.

> The Electronic Entertainment Expo (E3) is the big orgasmo-marketing event of the game industry, and thus, everyone in the industry expects to go every year. I know of one game company president who was unhappy

> with the progress of their game and told everyone they wouldn't have the usual company days off to attend. Then she changed her mind and was upset when one of the senior staff elected not to attend, anyway. So for future reference, is it important to attend E3 or not?

Use the Rhythm Method

Ideally, a project should run like it's on autopilot. Everyone by default should know what to do every day (if you have to have a meeting every morning to tell people what to do, then certainly this is not the case). When you're running a marathon, you don't have to tell your feet to take every step - it's muscle memory.

> Builds ideally should be automated and run overnight, with the results emailed to everyone on the team. On one game project, I manually ran the build every morning and prepared it for QA, with only a handful of hiccups for an entire year, but still toward the end of the project people kept asking me if there was going to be a new build that day. So the last couple of months I ended up manually emailing build notices every morning, also. I would have saved myself some grief if I had figured out how to automate the process earlier in the development cycle. That might have helped me avoid the problem of management asking me for new builds at various points during the day, which was not only annoying (imagine asking for a paycheck whenever you felt you needed one), but kept me from doing other work.

Stick with the Plan

If you have a reasonable plan, stay with it. It's hard enough to get everyone on the same page in the first place. Tweaking your plan or worse yet, changing directions in midstream, will impose some overhead and delay from the "context switch" and lower confidence in the project.

> Shortly after I joined a video game project, my group was diverted from their planned tasks to throw together a demo disc. This was not surprisingly a messy patchwork process, but once it was nearly complete, management changed their minds and put us back on the original schedule, albeit one month behind. The demo disc was resuscitated a few months later, but by then it was old code nearly impossible to debug.

There are times when a plan is failing and changes are necessary to have any chance of completing the project. But in those cases, you should be able to identify clearly the points of failure and be able to justify that the next plan avoids those problems, and more importantly, avoids the pitfalls in reasoning that led to the first plan. Otherwise, there is no reason to believe that at some point during execution of the new plan that you won't change your mind again.

Stop Pulling the Fire Alarm

Left to their own devices, management will often manufacture emergencies. Sometimes it's a conspiracy - one CEO will call a vendor CEO to complain and demand instant action. Or maybe Star Trek was on TV the other night, and the company president wants to be like Captain Picard ("Make it so!"). In either case, after raising some dust and getting some crunch time work, the bosses feel mightily pleased with themselves.

But an emergency indicates a failure in planning, and repeated emergencies will cause your staff to either leave or treat the emergencies as a joke (chances are you'll have both results - your more motivated staffers will leave and the less active ones will stay). Scheduled crunch times will have the same negative effects and add high risk to your project - if the crunch time is not as productive as expected, what are you going to do then? Double-crunch time? Try to understand - a smooth project is a successful project.

You're the Boss

You're the boss, so there's no reason to pretend you aren't.

Don't Ask, Don't Tell

It's not a democracy, so there's no point in soliciting input that you don't really want it.

> I worked for one manager who would ask everyone for their opinions and then get mad when someone disagreed with her. The only rationale I could see is that she would selectively bring up those cases when someone agreed with her to buttress her decisions later. But it would have saved a lot of time and argument if she had just made the decisions she was going to make anyway.

Of course, you shouldn't ignore good input from employees - just don't ask for an opinion if you don't want it. The most useless area to ask about is your own performance. It's like asking "Do these jeans make me look fat?"

> I once got into a dispute with a project lead who had made changes to my code when I was on vacation. He was unable to tell me exactly what those changes were, so I refused to integrate them. Offended, he asked me, "Don't you trust me to make those changes?" Now, why did he have to ask that?

Stop Whining

Yes, being the boss is hard. But no one cares.

> One of my more pathologically self-absorbed bosses practiced management by self-pity: "I never would have started this company if I'd known how hard this is." "Do you want to trade places?" "I know your point of view, but you don't know mine." "I kept all my promises and everyone else broke theirs." Believe me, it gets old after a few months, not to mention several years. This person even expected her employees to feel sorry for her whenever she terminated one of them.

Everyone is guilty to of some rationalization and self-delusion, but people who actually want to boss others around tend to be guiltier than most. (Hint: if you spend a lot of time thinking about how you're a good guy, you're probably not)

> One abrasive company president often complained to me how she'd be treated differently if she was a man. A departing employee agreed, observing "if she was a guy I would have punched her by now".

I enjoy working with managers who frankly discuss problems, but there's a difference between entertaining gripes and whining self-pity. If being in charge is really so unpleasant, quit and do something useful. If you feel unappreciated, tell it to your therapist. And if you're really narcissistic - well, you probably don't realize this section is about you.

Pick Something

Real leadership involves making the tough decisions. If you can't make those decisions, then, as I rather bluntly told a manager once - what do we need you for? The first choices to make are in prioritizing, and everything has to be prioritized - features, bug fixes, delivery dates, customers, hiring, buying equipment, and so on.

> While porting a Unix application to Windows, the president of my company told me the number one priority was to get the PC product out. When I asked him what about the Unix product, he thought a bit and said, "that's the highest priority, too". Thanks a lot, that's really useful.

Understandably, managers like to keep their options open. Sometimes this manifests as a refusal to make decisions early in the project, or a convenient amnesia regarding past decisions later in the project (you know this is going to happen when decisions are not written down). This strategy is as rewarding as setting up your retirement account just before your retire.

Lead By Example

No matter how well-written the employee handbook or how well-produced the corporate video, it is management behavior that sets the tone of the corporate culture. "Do as I say, not as I do" works great if you're satisfied with a mediocre organization or if you rule by fear. But if you want a high-quality group and you're not Stalin, you have to lead by example:

Be Realistic

The biggest management disease is wishful thinking.

> One Vader-like CEO intoned to me "This is the last time. You just have to take my word for it", when one of our helpful integration engineers insisted we had to deliver new features immediately after an official release. That assurance was as convincing as "this is my last drink". And it turned out to be just as valid.

Do the right thing now.

Be Responsible

If there's one common skill among up-and-coming managers, it's self-promotion. And self-preservation (well, that's two, but they come as a package).

> While thrashing around in a seemingly-doomed defense simulation project, I noticed the head of the department stayed a healthy distance away from the lab, for several months. But just a week before the deadline, when it looked like we were actually going to pull this project off, he showed up in the lab glowing with enthusiasm and optimism.

If you expect your people to put themselves on the line, you have to stick you neck out a little, too.

> One of my more annoying bosses liked to go around telling people "I'm holding you responsible" but whined "Now everyone's going to blame me" whenever she screwed up.

Everyone makes mistakes. At least if you 'fess up, you'll have the respect of your employees, and maybe, just maybe, they'll take responsibility for their work, too. It may not be the best corporate politics, and maybe you'll have to adjust your self image, but it's good leadership.

> I worked for one director of engineering who readily admitted her mistakes although she could easily have blamed all problems on our dysfunctional software development group. When a rogue engineer pushed ahead with his own release (and his own agenda, to curry favor with the client) independently of the rest of the project, creating more work and more grief for everyone else, this director admitted almost immediately that it was a bad decision on her part to let him do it.

> On a game project where it turned out there was an embarrassing flaw in the demo release. the director of software could easily have blamed a number of parties, ranging from employees to vendors, but he simply stated that it was a collective failure, we had to focus on a solution and fix the process to avoid similar problems in the future.

If you make it clear you've learned from experience, maybe everyone else will. They'll follow you with more confidence, instead of fatalism. And if you accept responsibility for mistakes matter-of-factly, instead of in some torturous sequence of denial and blame, then maybe everyone else will do the same and get on with the real work.

> One of my bosses seemed to expect a Nobel Prize every time she admitted to a mistake. Months later, it would be the subject of lore - "Remember I took responsibility for that? Remember?"

I've seen some leadership advice saying you should never apologize. I don't really agree with that, but an insincere apology is worse than none at all.

> One of my favorites pseudo-apologies: "It's my fault for hiring that person." A feeble show of responsibility while directing the blame at someone else.

Be Professional

Professionalism is underrated. Companies sometimes brag about how much fun they have and how they're one big happy family, but parties can only last so long, and how many families actually want to work together?

> I had one boss who liked to admonish employees to "please be professional". Which is one of the more annoying things one can say, like "don't be an ass". It was also tinged with irony, since her management repertoire included whining, yelling, foot-stomping, tossing items on employee's desks and terminating conversations by abruptly turning around and leaving.

You shouldn't have to like all your coworkers to have a decent job. (That would really limit your options) As with politeness, professionalism can make bad situations tolerable and make good situations last longer.

> The aforementioned employer told me she didn't care if departing employees had resentful feelings about their jobs, and then in the same conversation she wondered how to keep ex-employees from spreading a bad reputation for her company. Apparently, she didn't see the connection.

This is where insincerity is a good thing. Even totally sociopathic employers, if they're smart, will be smooth when easing employees out the door. "Glad to have you, sorry it didn't work out."

Be Fair

Don't be fooled by seeing all those warm bodies in the cubicles staying late in the day - they're not necessarily working (the Internet is a wonderful thing), and there's sure to be a stampede for the exit once the boss's car leaves the parking lot.

But employees might conscientiously put in the effort if they see that effort from you.

> On one tightly-scheduled porting project, I was responsible for the bulk of the code and consequently spent the first few months working eighty-hours weeks. I didn't get much assistance from outside the group, but I was greatly appreciative when the other members witnessed my long hours and offered to help in any way they could.

In particular, if you expect people to go above and beyond the call of duty and give up their personal time, you'd better show you're willing to make the same sacrifice.

> I've been on more than one project where the manager, just before going to the gym, would tell everyone to stay late at work. One manager asked me to wait for her in the office after hours until her aerobics class finished -

> upon return she ignored me another half hour until I finally asked what she wanted to talk about. "I just wanted to tell you you're doing a good job." Wow, that was worth skipping dinner.

The message - my time is more important than yours. Now get back to work.

Minimizing Meetings

> "What have here is a failure to communicate." - *Cool Hand Luke*

Avoid These Meetings

I'll take a strong stand on meetings right off the bat - meetings are evil, and not even a necessary evil.

In my first job out of college, the weekly meetings were painful experiences, filled with senior blowhards taking credit for work done by junior staff. After travelling on business one week, I enjoyed skipping the meeting so much I just stopped attending (shades of Office Space!). Apparently I didn't miss anything, I got more work done, and no one bothered me about it.

Most of my subsequent workplaces weren't nearly as dysfunctional, but still the meetings weren't much more useful. Automatic, repetitive, pointless - the worst ones are the typical ones:

The Weekly

A regular weekly group meeting is a traditional part of work life, like the coffee break. But the hours add up - say you just have one weekly departmental meeting involving forty people - that's costing you one person on the payroll. And do you really need to herd everyone into a room to get them to communicate? To share body odors and a group nap, yes, but after all, they are supposed to be working and communicating with each other during the rest of the week.

The most ridiculous weekly meeting I've witnessed (and there's heavy competition) was held by a manager who had just one person in his group besides himself. And they shared an office. Every Friday at 9am, the manager would turn around in his swivel chair and say to his officemate, "Ready for our meeting?"

Sometimes meetings are held just to remind everyone who's the boss. If that's what it takes, you've got other problems.

The Standup

I'm not a fan of scrum. And I don't mean rugby. The thing I hate about the variant of Agile Development called Scrum is its defining characteristic, the daily standup meeting.

The daily standup is a reaction to the traditional weekly meeting, but it's no better. Rather than waste an hour of everyone's time once a week, it takes ten minutes of everyone's time every day. Factor in the context-switch time of breaking off work and then returning to work, it's not a real time-saver.

The daily meetings are also a ritualization of the what-are-you-doing-right-now school of management, which is not exactly goal-oriented. It's irritating enough to have a manager go around a table once a week and ask "what are you working on?" but to go through that every day is like being in kindergarten. And when I did sit (stand) through such daily meetings, I still had middle management come by later in the day to ask "what are you working on?" (I always wanted to ask them the same question)

The Pep Rally

I am forever grateful to the high school math teacher who let us spirit-impaired nerds hide out in the computer lab during pep rallies. They were a welcome break from class, but the fact that hip-waggling cheerleaders stirring up support for our football team (despite its two consecutive years of losses) warranted time taken out of our studies gives some indication why this country isn't preeminent in primary education.

Rah-rah corporate meetings also provide a break from work, minus the cheerleaders. They're really an opportunity for the CEO to flex his CEO muscles - an exercise in power management. Sometimes the hypnosis works and employees fall under the sway of the exalted leader, and sometimes they just find the guy annoying, but I've never seen employees come away actually more encouraged and determined to work harder.

The Come to Jesus Meeting

I've heard these referred to as "come to Jesus" meetings, but I think of them as panic meetings, where management realizes, belatedly, that things are not as they should be, and calls an all-hands meeting to tell everyone to shape up.

I've never seen anyone come out of these meetings energized to work. Usually, they go off to polish their resumes.

The Group Hug

At one point during my first (and worst) job, a self-appointed morale coordinator who apparently had nothing better to do first issued Briggs-Meyer tests (we just handed the forms to our friends so we all got the personality profiles we wanted), and then when team unity failed to result, set up an off-site two-day team-building session, down the street from our office. We held hands, took turns "confessing" how we could have done a better job (mostly phrased as "I could have done a better job in helping that guy do his job", except for the fundamentalist Baptist who refused to confess because he wasn't Catholic).

Here's an idea for a team-building exercise: finish the project.

A Few Good Meetings

Despite my allergic reaction to meetings, I do believe a few can be useful, and they may be appreciated more for their rarity.

The Kickoff

The first useful team meeting is the project kickoff meeting. It's important to make sure everyone is on the same page from the very beginning.

On my first project where I was a lead, I thought the project goals were pretty clear. But after a few months, the team members actually requested a meeting. After that, no one asked for another meeting, so either it was satisfactory or totally useless, but I do wish I held that meeting at the beginning.

The Countdown

There is one part of a project where I do think it's important to for everyone to get together and know exactly what's going on - that's the final days before a project release. The code should be frozen and undergoing extensive testing at this point - every day there should be a meeting to go over the latest test results and decide what to do about them.

The Postmortem

Memories fade quickly, so start off the new project cycle with a meeting to discuss lessons learned. And anyone who feels abused by the past project is likely to be annoyed enough to honestly air their views.

However, even meeting-happy managers I've worked for have been reluctant to hold postmortem meetings, ostensibly to avoid finger-pointing (probably at themselves), but I suspect also because the point of a postmortem meeting is to take a hard, realistic look at the process and make any necessary changes. And no one wants to do that.

I did participate on one video game project that held a few postmortem meetings after milestone deliveries, and the sessions were useful in refining our process. Too bad all that got thrown out the window at crunch time.

Use Technology

Technology has changed how we do nearly everything - entertainment, travel, shopping, managing our finances. So why should we hold meetings the same old way like cavemen grouped around a fire? Some technologies, like videoconferencing, just make it easier to have more meetings, much as some technologies create more paper instead of advancing us toward the paperless office. But technologies that we use to streamline the rest of our lives, standard stuff like email and web browsing, can minimize the number and duration of meetings.

Status Reports

The most important email everyone can issue is the the weekly email report. Rather than wait for a meeting to go around and ask everyone for a status report (the most painfully boring type of meeting), the reports instead can be composed and emailed by each member of the team before the meeting - ideally, the day before the meeting. Then everyone can put some time and thought into the report, instead of dredging it up on the fly in the meeting, and everyone can spend some time digesting the reports before the meeting and consider issues to bring up at the meeting.

In all cases where I've seen the weekly email report initiated, the time spent in the weekly meeting easily halved, and the content of the meeting was more focused. Moreover, email reports serve as written records that can be easily referenced, instead of "I'm sure I mentioned this at the meeting two weeks ago...."

Whether or not the weekly report happens before the meeting or during, the content should not be "what I did this week". (I had one coworker recount his trip to the dentist, and that was one of his more productive weeks) It's a "status" report, not a "what have you been doing" report. If you're more concerned about how people are spending every minute than what actually gets done, then you're better off installing monitoring software at every workstation. Asking people what they've been doing sends a message that you care more about looking busy than what's actually getting done.

The report should contain information that is actually useful and gives everyone else on the team a feeling that they know what's going on. And, to put it crassly, it's also a CYA device for developers who feel their contributions and concerns may be glossed over by their managers. (To put it less crassly, think of it as an implementation of the open-door policy that companies like to brag about) The reports should include:

1. Changes made to the system that everyone else should see. (For source code, these are changes committed to the source code repository and can be verified by others, not in-progress code that is on the developer's machine).
2. Dependencies that the developer is waiting on - hardware, test cases from QA, code from other developers, decisions from management.
3. Problems that require attention - current approach is not working out, the developer is overloaded, progress is slow and target date will not be reached without more resources or trimmed features.

Everyone should issue a weekly report, not just the programmers. There are a lot of tools for monitoring software development - keeping track of marketing, sales and management activity is another matter. If every department is made more "transparent", that can counter the typical perception from engineers that everyone else is on a coffee break while they're under the gun. And if it turns out some people don't have anything to report (or are reporting the same thing for weeks on end) then now you know you've got a situation.

The Intranet

All project information should be accessible on a local intranet server. These days, it's not uncommon to find a wiki set up on the intranet - this makes it easy for project members to add and correct project information. And then there's no excuse for anyone to not understand what's going on in the project.

Bad Technology

On the other hand, say no to instant messaging and any other form of communication that facilitates interruption. Whereas use of email promotes thought and clarity in expression and response, instant messaging caters to the short attention span.

I had one manager who couldn't stop reading and writing her IM even while I was standing next to her giving a status report. Great example for the staff!

Some managers don't like their employees wasting time instant messaging with the outside world, but internal IM is even more damaging - coworkers and managers will get in the bad habit of expecting instant responses to off-the-cuff questions, instead of well thought out answers to valid questions.

Stay Focused

Have an Agenda

It's easier to get decisions made in a meeting if you plan the meeting that way.

There are few events more uninteresting than going around the room asking for each individual to give a report. Once you get rid of the round-the-table what-have-you-done routine, then it becomes clearer that you should have an agenda (and if you can't think of one, then there's no point in having the meeting).

Certainly, a standard part of the weekly meeting agenda should be to address any concerns that have been raised in the weekly reports.

Release the Hostages

Now that you have an agenda for your meeting, stick to it. This doesn't mean slavish devotion to a list or formula - just remember why you're there.

Some people take advantage of having a captive audience in meetings. One company president would regale us with a review of last weekend's movie she saw or game she played, or relate some formative event in her life. Another producer would spend the beginning of the meeting telling bawdy stories and insulting the French, then snap at everyone else to stay focussed.

Remember, it's not a social gathering. Save the fraternizing for happy hour.

Start on Time

The first aspect of a meeting is, obviously, starting the meeting. Adhering to a strict start time sets the right tone - get going, get to the high points, and get this meeting over with so we can get back to work. On the other hand, inconsistent start times not only waste time as people sit around waiting for the meeting to start, it sends a subtle message that time is not valuable, or that some people's time is not as valuable as others.

I had one manager who would repeatedly postpone a meeting to attend to various things, and then she would complain about how she had to round up everyone for the meeting when it actually took place. Not only was this was just annoying, but it meant no one could effectively plan their work around the meeting - you would either get interrupted right in the middle of trying to figure something out or you'd be twiddling your thumbs waiting for the damned thing to happen. Not a good way to promote time management discipline in a project that was already challenged with a tough schedule.

End on Time

The reasons for starting meetings on time also apply to ending them on time.

The best regular meetings I ever attended were held by an MIT professor for his research group who always ended his weekly meetings on time. Even if there were more items that could have been discussed, the professor still adjourned the meeting and we tabled the discussion for next time. I left that meeting never feeling that it dragged on, and on at least a few occasions I felt it was productive and worth the time!

Ending early is OK, but letting a meeting go over the prearranged time is the same as allowing a schedule overrun. If you can't keep a meeting on schedule, what hope is there for keeping the project on schedule?

Write it Down

Have you been to meetings that seemed like the movie Groundhog Day? Same day of the week, same time, same topics, only it feels like you're the only one who remembers going through all of this before. ("Vampire topics" that repeatedly come back to life, as one of my friends termed them) Or perhaps you're one of the blissfully bringing up the same topic again and again.

I participated in a two-day annual sales meeting for a startup company where supposedly we discussed our product plans and marketing strategies. You'd think in a startup with one product and only a handful of sales and marketing folks, someone would remember whatever was decided at that meeting, but I found myself repeating peevishly for months, "We discussed this at the meeting!".

There's not much point to a meeting where decisions made are not remembered. So make sure everything of note in the meeting is written down and distributed to everyone who should know what's going on. The intranet wiki is a great place for that - then you have a history of meeting results that you can refer to.

The problem isn't always due to short-term memories (possibly salespeople only remember as far back as their last commissions). Sometimes memory is selective, especially among management.

On a video game project where I was the sole programmer for a specific platform, the president of the company asked if we could develop an interactive previewer for that package. I responded that we didn't have the manpower to do that and stay on schedule - if it was important, we needed a tools programmer. Yet that topic kept coming up - either she kept forgetting my answer or thought I would eventually forget.

And if you do have someone record the meeting results, make sure the recorder can do so reliably.

Case in point: one of my coworkers who was regularly aggressive in pushing his own agenda was all too eager to record our weekly meeting results. Somehow, the meeting minutes always indicated his opinions had prevailed.

Having a fictitious record of a meeting is even worse than having no record.

It's the User Interface, Stupid

"It's a Unix system! I know this!" - Lex, recognizing a 3D desktop interface, in *Jurassic Park*

You Have to Care

Anyone can create a decent user interface. Designing reasonable user interfaces is not rocket science. It's not even computer science. I know there are plenty of researchers dedicated to the study of effective user interfaces and their design, but bad user interfaces occur not due to lack of training in cognitive science - bad user interfaces happen because people don't try.

> If you pay me a visit at my condo complex, it might take you a while to find my name on the phone number list pasted by the entry intercom. It's not listed alphabetically by name - instead, it's sorted by the three-digit phone code. So if you knew my phone number already, you could look it up easily! One might suspect that the staff at our condo management company was just sticking to an unfriendly but logical system, but there are a few names, including mine, tacked on at the end of the list in seemingly random order. Those names happened to be the ones the management staff forgot to include the first time around.
>
> Now, it's not like the names are engraved in stone. The page was obviously printed with a word processor. It's not that hard to move the text cursor when you're typing, and it's not hard (I hope) to alphabetize, especially when the list includes less than twenty numbers. This is just another case of someone not caring enough to do a good job or think about how to do a good job.

Designing a good user interface isn't always easy, but anyone can identify a bad one. We deal with interfaces all the time, from web sites to ATM machines to microwave ovens, and we have an idea what interfaces we like and which ones we don't. Even those of a contrary nature who delight in presenting puzzles, or those of an apathetic nature who just don't care, still have strong opinions on the interfaces they have to deal with.

That said, not everyone is suited to design an interface for everyone else. For example, programmers who like command-line interfaces and don't see the point of windowed desktops are probably not going to make very good GUI's. But it's a bad rap to attribute bad user interfaces as the work engineers who design products only engineers can use. Forget that leftbrain/right-brain folderol. Code API's, whether internal or external, are a form of user interface - good ones are a delight to use and bad ones are painful to use, causing errors, delay and aggravation. (If you don't like Microsoft GUI's, try looking at their API's) And a "creative" graphic designer might design something that would look great in a Powerpoint presentation but completely mystify users.

The best interfaces appear in labors of love that the creators want to take home with them. The worst products are those implemented with just enough effort to satisfy checklists from the marketing department.

Be Consistent

The most important rule in user interface design is consistency. Just as drivers rely on consistent conventions with the shape and content or road signs, the user can't navigate the application effectively if it involves constant trial and error.

> I see this problem in various Windows applications (and Windows itself), when many, but not all, operations on an item are available from a right-click menu, and some are only available from one of the menubar menus. This results in time wasted when the desired function is not immediately found, and an obviously less trustworthy user interface model.
>
> A common inconsistency in computer and video games is the lack of consistency between options in the front end (at the beginning of the game) and those provided by the in-game menu. For example, if audio settings are available in-game, why should you have to exit to the front end menu to change video?

It's better for the interface to be awkward in a consistent manner than to be inconsistently intuitive in some areas and different in others.

> The Windows recycle bin is a great safety feature (although not an original idea, of course). To really delete a file you have to delete and then really delete by emptying the trash bin. On the other hand, as I discovered to my chagrin on one project, the same delete from a network-shared disk is only a one step process.

Microsoft is an easy target, but Apple isn't guiltless, either.

> I was one of the original adoring Mac fans, but the insistence (was it Steve Jobs?) on a one-button mouse was misguided. Since the resulting hack - double-clicking - was copied blindly by Microsoft along with everything else, I have spent hours explaining with torturous logic to my condo neighbors (they figured out I was a computer guy) why they're supposed to double-click on desktop icons but single-click buttons and web links, and forget control-click, shift-click, option-click or (Mac) command-click. And watch out for those laptop touchpads that turn lingering thumbs into

> double-clicks or mouse drags! Even the MacOSX desktop can't make up it's mind - double-click opens files in the Finder, on the Desktop and in chooser dialogs, but the Dock and the System Preferences respond to single-clicks. More than once I've accidently changed my screen resolution by double-clicking the Display applet (the first click opens it, the second chooses an annoyingly low resolution setting)

Inconsistencies in API design slow down programmers and are sometimes downright dangerous.

> I recently helped a client resuscitate a legacy Windows 98 product and get it running on Windows XP. Tracking down some odd artifacts in the GUI, I was horrified to discover that all of the MFC classes took an attribute mask in the same parameter position except one, and the previous engineers on the product hadn't noticed the exceptional case because the bits they passed as the wrong argument just miraculously happened to work - seven years ago.

Make It Clear

Clarity seems like an obvious goal in user interface design, yet how many programs allow you to sit down and just get started?

> I thought after the whole hanging chad business that the new electronic voting systems would at least have simple user interfaces. And the ones I've encountered at my local polling location do have a very simple controller - a clickable track dial for moving the cursor around the screen and selecting answers. However, the system allows you to position the cursor anywhere on the screen, including the question fields, so it's not clear what you're supposed to select, and if you do select a field that's not selectable, well, nothing happens.

API's also can suffer from tack-on-itis.

> If you program with the Microsoft Win32 API, you have to remember to use the newer versions of some functions that are simply extended with an "Ex" suffix and additional arguments, e.g. CreateWindowEx instead of CreateWindow. Try using Win32 without constantly referring to the documentation. Just try it.

Make It Easy

The one and only time I tried to order from a certain MacDonald's near Johns Hopkins University in Baltimore, the cashier responded "We don't have any right now." After trying a couple of other items, she finally informed me that they weren't serving lunch, yet. But as I ordered a McMuffin the minute hand advanced from 10:59 to 11 AM and the cashier duly informed me that breakfast was no longer being served.

User interfaces can be bureaucratic, too. Some force you to jump through a specific sequence of hoops, and sometimes you have to start all over if you get one thing wrong. Voice mail is pretty much designed to do this. But often this results from implementors taking the least-effort approach and just tacking on functionality.

> High-end 3D content creation systems tend to have feature-rich interfaces - I worked on one that just chained together sequences of generic popups because that was the easiest way to program it. To use a texture, bring up a file chooser to select the image file, a color chooser for each of several shading options, and popup numeric entry dialog for each numeric option. Contrast this with the Macintosh control panel (old or new) - everything in one dialog.

Put the related options together on one dialog, and don't make the user leave until done. And think about whether you even really need a dialog.

> In Windows Explorer, you have to right click on a drive icon and select the Properties dialog to see how much space is available. In the Macintosh Finder, it's always displayed at the bottom of the window.

Jumping through hoops on a desktop application is annoying enough, but on web sites it's excruciating.

> I've found travel planning on the American Airlines reservation web site terribly annoying if I'm trying browse. Type in your travel start and return dates, hit Submit, and if you don't find something you like, you have to start all over. If you don't "Search by Fare", the resulting list of flights doesn't show the respective fare, and you have to choose a flight and proceed to the reservation to see how much it'll cost.
> The sites for other airlines (Jet Blue and Southwest, for example) allow you to conveniently browse forward and backward a day at time to compare flights. Moreover, they immediately show results for surrounding days so there's a good chance you'll see all the options you want.

Obviously, it should be obvious what to do. If you have to explain it, there's room for improvement.

But Not Too Easy

On the other hand, there are some functions that you don't want to make too accessible.

> I worked on one rather large and flakey application that had a "Reset" button prominently displayed on the user interface. Users would click on that button as if it was an accelerator pedal anytime the application seemed unresponsive, i.e. anytime the user became impatient. The reset function was useful, but should have had either a confirmation step or been an option within a menu. As it was, I suspect the reset probably

> interrupted more operations than it recovered, and it surely made debugging more difficult.

Emergency and safety features, like fire extinguishers or the emergency break in your car, should be obvious in function but inconvenient enough to prevent accidental use. But in keeping users out of trouble, don't be patronizing.

> I worked on a chart display that had all kinds of unnecessary "features" that tried to avoid displaying anything that was not deemed presentable. For example, if the user zoomed out enough (using a scale slider) so that smaller items on the chart did not fully enclose their text labels, then those items simply disappeared from the chart. This was a bad solution to a non-existent problem - it's easy enough to zoom back in if the display is offending the user's esthetic sensibilities, and magically omitting data risks confusing or, worse, misleading the user.

Don't presume to tell users what they want to do or see. Just give them the tools to do it.

The Customer Is Always Wrong

Well, not really. But if customers really knew what they wanted, they could start their own company.

> I worked on a computer graphics tool that featured a somewhat atypical menu system - rather than go to a menu bar or an ever-present vertical list of all the menu options on the side of the screen (common among our high-end competitors), our users would shift-click inside our 3D modelling window to get a popup with cascading menus. When we took a survey among potential customers, they said they wanted a vertical side menu.
>
> Of course we implemented the side menu, but that was a lot slower for professional users who don't want to drag the mouse across the screen every time they need select a new operation or mode, so we had another mouse click warp the mouse the side menu - the type of design decision that only makes sense in the context of a previous bad decision. And then of course the heavily populated side menu didn't fit on the screen when we ported the product from the high-end Silicon Graphics workstations to Windows NT machines. And guess what - anyone who actually bought and used the product for serious work used the original menus.

If it's as easy as just asking customers or your salespeople, then hire them to do your job. Relying on user surveys, focus groups, marketing checklists and the like is a cop out. There's no substitute for observation, analysis and innovation. And you don't need huge resources to get it done.

> I spent one project listening to a user interface programmer complain that we needed user testing. Meanwhile, he was oblivious or dismissive of every stumble, inconvenience and point of confusion encountered by the testers and project manager when trying out the product.

You don't need to make user testing a big formal deal. Just do it. Watch, listen, learn.

The Computer Is Always Right

User interface specialists often ridicule programmers for presenting a user-interface that models what the program is really doing instead of a simpler user model. But they are at best half-right. Hiding complexity is usually a good thing, but presenting a user model that does not match the the application's internal model is a recipe for trouble.

> When I was a user-interface programmer for a large video game project, the game designers came to me with all kinds of game features that they just assumed involved cosmetic tweaks to the HUD, since they just considered how those features would look. More often than not, they involved fundamental changes to the game engine.

If the application model is too difficult to use, or just plain wrong, then that's what needs to be fixed.

It's Not Just a Pretty Face

User interfaces are getting better looking all the time, but looks aren't everything.

> I once worked for a neat freak who went around the office at night arbitrarily tossing anything she found on desktops into drawers. One book went missing for months. Worse, she instructed her lackeys to wire up the computers so it was nice and tidy, and utterly inconvenient. The test staff had to move desks in order to switch test equipment, and I spent an afternoon testing video game memory functions by repeatedly crouching and blindly reaching behind a box under the desk, because gosh, it would have looked too ugly to have those slots facing forward!

Good looks make a good first impression, but that's all you're getting if the interface is painful to use and everything is hard to find.

> I worked for one manager who had a minor in visual arts and insisted on designing the user interface himself, mainly by drawing pictures. That's not bad in itself, and the pictures looked nice, but he seemed to expect that to be the whole process. There was no description of the user flow, not much thought to consistency, user model, or grouping controls by function rather than appearance. And despite this being a Mac app, no reference (or reading) or the Apple Human Interface Guidelines, which is the industry gold standard.

Artistic skill is useful (I'm afraid I can't design an icon to save my life), but user interface design is about analysis, empathy, and diligent research. It may not be rocket science, but it is engineering.

It's Not Just the Application

User interface design is about user experience, and the application interface is just one part of that.

> Apple gets it right. Buying my Macbook Pro at the Apple Store was like buying a luxury car. I spent some time gazing at it in the showroom (the Apple Store), purchased it in a few minutes from an Apple Store rep who immediately retrieved it from the back and handed it to me, carried it out in a nifty box with handle, carefully unfolded the quality cloth that wrapped the laptop, and zipped through the setup process. And now, passersby at Starbucks often comment how cool my laptop looks.
>
> Contrast this with my previous computer purchases. Every time I bought a computer from Fry's, it turned out they gave me a previously-returned box (in the case of a Powerbook, the registration had been partially filled out and the MacOS install was only halfway complete - as it consisted of a dozen floppy disks, I guess the previous purchaser got tired of swapping floppies). Best Buy aggressively pushed an extended care warranty on me, left me in line for fifteen minutes to complete the purchase, and somehow I got billed by Microsoft for their Internet service (I wasn't the only one - I discovered years later that I missed out on a class action lawsuit). At CompUSA I watched a sales associate hunt-and-peck an old terminal for fifteen minutes so he could print out the paperwork on an equally old dot-matrix printer. Not quite as bad as the buying experiences for both my Toyotas (nice cars, slimy salesmen), but I don't have to upgrade my car every couple of years.

It doesn't matter how well you design your application if the user is turned off before even using the product. Whenever possible, I start all of my projects in designing the installer. And don't forget support - all of that goodwill engendered by smooth installation and a nice interface can be cancelled out by one lousy call center.

It's Not Just For Computers

While it's easy to complain about the state of computer user interfaces, and it's all warranted, at least there are plenty of people complaining about it. If only that much attention was paid to everyday things.

> Take my bills (please). Like GUI's, they keep getting glitzier, as my phone and utility providers proudly announce their "new look". But I still have bills that are folded and creased just far enough that when I detach the

tear-out stub to include with the check, they don't fit in their envelopes without folding them down a quarter inch. My utility bill doesn't list the account number on the stub, so I'll detach it, fill in the balance paid, start writing the check, and realize I have to go back to the other half to find the account number.

My doctor's bill is worse - it doesn't even show the balance due on the stub. And while the main part of the bill shows the balance (of course), it only lists account changes in the month the bill was issued - any other charges made since my last visit to the doctor are included in the balance but will never, ever be listed anywhere! My question, why bother listing any charges at all?

The Art of the Schedule

"A canna' change the laws of physics. I've got to have thirty minutes." - Scotty, in *The Naked Time*

What is a Schedule?

On many projects I've been told there is no schedule - either as a complaint ("We don't have a schedule!") or boast ("We don't have a schedule!"). But there is always a schedule, and you'll find out what it was when you don't meet it.

During an interview with a prospective client, I was told "we're not going to release this product until it's ready." Soon after I started working there, the president of the company called an all-hands "come to Jesus" company meeting, where he yelled at everyone for not having the product finished several months ago.

"We won't release the product until it's ready" - famous last words. Only well-funded and self-funded companies can really say it, mean it, and do it, and you usually hear this said after a product misses it's original release date. So how do you make schedules you can meet and meet the schedules that you make?

Before blathering on about effective project scheduling, let's be clear about the definition of a schedule. A schedule is an expectation of delivery dates - most importantly the product release, but also key demos, milestone deliveries to customers, promised interim release dates. Don't confuse the schedule with a glorified task list, a daily to-do list, or lines of code written per day.

The management of a startup company that was cruising in research and development was obviously getting restless, so I suggested we establish a schedule. This inspired the project manager to establish a day-to-day task list for each engineer during the next two weeks. When I clarified that I really meant "when do we have to have something you're going to show",

> upper management commiserated and then informed us that we had to show a demo to investors at the company launch party in a few weeks! The micromanagement list went out the window and we scrambled to implement the key features needed for demo. (We came close, but I can't say it was a success - we got the demo running just as the party was winding down, and the funding fell through.)

A project's life or death hinges on getting good demos in the premier trade shows, satisfying contractual deliveries to clients, and timely appearance on the market. The entire project revolves around these expectations, so the first job in scheduling is to figure out these expectations and make them and their assumptions explicit.

It's a Deadline, Not a Guideline

After understanding what a schedule is, the next important thing is to take it seriously. Know your schedule, and make sure everyone on your project knows it.

> I was astounded on one project when casually mentioned our recent product release and a fellow engineer on the project didn't know what I was talking about. But I couldn't blame him for that head-in-the-sand mentality - our management would announce "stop coding, we're making a release today" and then justify it with "well, the beta release date was last week."

Any computer scientist should be able to tell you that constraints are a good thing when it comes to problem-solving - they narrow down what you can and can't do. Take advantage of the fixed milestones in your schedule, like trade shows. Milestones that you can't shuffle around in self-denial are useful, like the scale that won't lie about your weight.

Every missed deadline means the next one will be taken that much less seriously. And every project that runs overtime means the next project will start and end that much later.

> In a surreal moment, just a few weeks before a product release, timed to coincide with the premier industry trade show, and right after I'd dissuaded the company president from demanding new features by pointing at the bug list, a well-intentioned marketing manager suggested to me "why don't we postpone the release by a month so you guys have more time to work on it?" Setting aside the trade show, which should have concerned him more than me, we had a supplementary feature release scheduled just three months after that, and then big upgrade a few months later. Delaying the release by a month would have just delayed all the other releases. And it wouldn't have saved us any work - we would just have been behind by one release month.

Stick with the schedule. There's always a next time, if you get this one done on time. And while delivering late is criminal, there's no sin in delivering early.

> For some reason, even when given the option to either a) release the product early or b) take the extra time to test the product, management usually elects option c) squeeze in a new feature that jeopardizes the the product and the schedule. (In fairness, engineers are often complicit - "sure, it'll just take an afternoon"). The one time I've seen a product delivered early took place after the president of the company gave the development manager a fake release date. I can't imagine that working more than once, though.

It's Macro, Not Micro

When I shop for groceries, I know it's not going to take more than half an hour, and I'll spend about forty dollars (sixty, if I'm hungry). I know this because I've bought groceries before, and that's generally how long and how much it takes.

Now, if I plan my grocery expedition by listing all the items I think I'm going to buy, estimating the unit price of each item, and predicting how much time it will take to park the car, locate each item, stand in the checkout line, and unload each item, my schedule will be off by the time I reach the first aisle. It's unlikely that my predicted individual prices will be on the money, so to speak, but it would be a waste of time to recalculate my schedule every time I find an item and its actual price. Even if I have a detailed grocery list written down, I'm only committed to the items I really need (can't postpone getting that cat litter) - I may purchase additional items that happen to be on sale and omit items that turn out to be too expensive or are unavailable.

The same goes for project scheduling. Estimate project durations based on what you know of previous projects - that's where experience comes in, and that's a big reason why senior engineers should get paid more, if they've learned anything. Estimating project durations by summing up individual tasks only works if you've worked on exactly the same kind of project before and know every single task and possible schedule deviations that could happen in that type of project. In which case, you'd already know how long the total thing took, so why not use that, anyway?

> I worked on one distributed simulation project where the manager's office wall was completely covered with a Microsoft Project schedule. I assume this included every task going into the project (and maybe tasks not going in). But this project actually had a very simple schedule - have the simulation ready for the exercise date in six months, or the exercise would proceed without us. Major milestones, like having the software components ready for integration, the network hardware up and running comfortably ahead of the exercise date, and even my security clearance so I could start work, took months longer than they should have. Yet we had our meetings where everyone reported that everything was going fine and more paper went on that scheduling wall. The final month of that project was an exercise in panic and triage.

Schedule the major milestones first - the final release, the beta and alpha releases, and the interim milestones like first QA release and trade show demos. These are your hard targets - everything else can, and will, be more malleable, as you find certain technologies are working out better than others, you have to deal with staff turnover, sick days and vacations. I personally don't feel it's useful to schedule tasks to a granularity smaller than a week, but regardless, don't let your micro-schedule drive your macro-schedule - tracking low-level tasks looks good on paper, but the feeling of control is illusory - it's no substitute for "macro" planning.

It's Not a Negotiation

I've worked with managers and clients who would ask for a time estimate on a task, and then respond to the estimate with "why should it take so long", or "can you get it done sooner?". There are several things wrong with this approach:

> If you're taking issue with an estimate, that indicates you already had a preconception of what the estimate would be.
>
> There's no reason to believe a revised estimate will be more reliable than the first one. On the contrary - it will probably be less valid.

Remember what a schedule is - a prediction. I have allowed myself in some cases to lowering time estimates in these "negotiations", but it didn't change how much time the task actually required. If, as a manager, you are asking for a lower time estimate, then you are saying the original estimate is wrong, and if you receive a revised prediction, whether it's more or less to your liking, you should ask yourself why the revision is more believable than the original prediction.

Likewise, as a client, you may want to bargain down on price, but bargaining down the time estimate will just backfire on you - the project will take as long as it takes.

That doesn't mean you cannot shorten the time for a project, but that means making reasoned tradeoff in features or resources. The right question to ask is "What can we do to get this project done earlier?"

Schedule Suboptimally

I used to have a terrible punctuality problem when I lived in Boston. Whenever an appointment time came around, I would tell myself I just need, say, fifteen minutes to drive down Memorial Drive along the Charles River to make a meeting around MIT. Invariably, after giving myself that fifteen minutes, I would still get started late, get bogged down in Boston traffic (sometimes I forgot to factor in the rush hour impedance). And then there was the time spent getting lost, if it was an unfamiliar meeting place, finding parking (certainly a high risk factor in Boston), and navigating the destination office building.

This planning could be charitably called "best-case" planning, but I think everyone would agree it's a stupid way to schedule, especially after missing more than one meeting (including job interviews) by a wide margin. Nowadays, I'm much better at this - partly because I moved to

California, but also because I do take in account how long these trips usually take, and I give myself some extra time in case traffic is slow or I have to stop for gas.

Software projects should be scheduled the same way. Look at how long it's taken your group to do similar projects and add some safety time to take in account things that might happen. You don't necessarily have to take in account worst-case scenarios - I don't factor in potential three-hour freeway stoppages when I plan my outings, but they do happen once in a while. But do consider carefully how much risk is acceptable, a question that is de rigeur in other aspects of engineering.

The First Schedule is Always Right

One consequence of making a distinction between the macro schedule and micro schedule is sticking to the initial macro schedule. The schedule you set down at the beginning of a project is most likely the correct schedule - you're basing the delivery date on market considerations, expected budget, staffing and technology capability, and the beta, alpha and interim milestones all fall from that. At this point, you're most free of wishful thinking influences - the schedule should be consistent with your experience from your own projects and others that this is a feasible schedule. Some things will take longer than expected, some shorter, but you'll arrive at the expected time.

> Three months into a a one year development schedule to port a 3D graphics application from Unix to Windows, I suffered several bouts of unwarranted optimism. The user interface system was basically up and running, and I told my boss I thought at this rate, I'd have the project done six months earlier. Fortunately, he didn't tell anyone. With Windows idiosyncracies, graphics card issues, tracking ongoing changes from the Unix product line, and battling a compiler that was also in a "pre-alpha" stage, it wasn't long before it was obvious we were exactly where the originally schedule said we'd be.

But no good deed goes unpunished. Once it appears your project won't be a disaster, everyone who was keeping a healthy distance away will suddenly show up to "help".

> Ten months into a year-long crunch project, at the beginning of which it was wisely decided to pare features to enable a release by the next industry show, my boss suddenly wanted to put those features back in. He went around me (he knew what I would say) and asked some of the more optimistic and eager-to-please engineers to put in the features. After several days effort, I put my foot down and explained we already had a hundred and fifty bugs marked by QA as critical, and we could not afford the time to put in new features. "Oh, I didn't know we had so many bugs," was his lame acquiescence.
>
> On a game project at a different company, the president tried to accelerate the project by promising game delivery two months ahead of schedule. It

seemed like things were going well, and finishing this project early would have let us get started on the next game, so I agreed, but once again, I was wrong. When we reached the revised deadline, we were just starting find all the bugs that had to be fixed, including extensive requirements by the console makers, whose approval we needed to distribute the game.
So we went back to the original schedule, but during the last week, once again, the president tried to advance the submission date by a couple of days. And once again, we didn't make it. Worse, for every attempt to finish early, there was an extra flurry of activity, extra builds and late nights, that were essentially wasted our energy and sapped the momentum of the team. When you're taking catnaps in your parked car during the day, you're probably not producing quality work (especially if you own a compact car)

There will always be at least one point in the project where it seems like things are going better than expected. It's not. If the project actually gets done early (but it won't), then ship it, and if you want to maximize the chances of that happening (but it won't), then focus on getting those requirements done and avoid other distractions, like additional features.

Get a Job

"Does that come with dental?" - *Grosse Pointe Blank*

Find Your Next Job Now

I've had twelve salaried jobs so far (nothing to brag about, I know - it probably indicates I'm pretty good at getting a job, but not so good at picking one) and I've been called a "short-timer" by many of my ex-coworkers. But in retrospect, it's apparent I've never left too soon (unless you count the severance pay that I missed). Even if you're reasonably content with your current job, you should be prepared if it becomes time to find a new one. Even if you have to settle for less just to pay the bills, you'll be ready when opportunities present themselves.

Know What You Like

Rather than waiting until a new job opportunity shows up and then rationalizing that it's a good move, recognize early on the type of work that interests you.

I made one of my worst career choices when moving to Silicon Valley. Moving to the center of the dot-com boom near its peak was a good move - taking a job offer because it was convenient was the mistake. With a little more effort I could have continued talks with AvantGo, a company that appeared to be filled with talent and developed mobile applications, one of my top interests. Instead, I went to a software startup that had just been acquired by PRI Automation, a supplier of semiconductor fab equipment

> company I knew and respected the CTO whom I had worked for previously, and they offered me a job almost immediately. After a wasted year of turmoil and corporate intrigue, during which I found the whole field of manufacturing completely boring (and the state of manufacturing software completely appalling), I found another job at a mobile internet startup called Neomar. That didn't last long, either, but it was a lot more fun, and we had a product out in six months.

Don't fall into the trap of just taking a job because it's convenient, or because you're dissatisfied with your current job and anything new sounds better.

Know What You Want

I have an uncle, a Silicon Valley executive, who has tried to impress upon me the importance of having a five-year career plan. I still prefer to wing it (and look where it's got me), but I do feel you should know the direction you want your career to go.

> My best career move was in leaving the defense contractor BBN for a small company developing commercial 3D software called Nichimen Graphics. BBN had a great technology history and culture, but I'd worked on corporate and government projects for my entire career, and, especially with the Internet boom starting, I wanted to work on products that you didn't need a security clearance to use. Nichimen didn't last long, but it turned out that I really did get more satisfaction from delivering shrink-wrapped packages to real consumers, and small companies moved at a pace that I enjoyed.

Figure the Odds

My advice - don't take a new job unless it offers at least two distinct things that appeal to you.

> I once asked a prospective employer what I would get out of the job. The only reason she could come up - "You'll learn a lot." And I did - mostly, that I shouldn't have taken the job.

It's a matter of risk management - if you change jobs based on one reason, it may well turn out that rationale was invalid. But if you have more than one reason for taking the job (and being unhappy with your previous job doesn't count), then there's a decent chance you won't feel like you made a terrible mistake.

Peer-to-Peer Networking

Recruiters writing job-advice columns love to tell you how you should network by calling up all everyone you know when you need a job. But that's not networking, that's panhandling. It's what recruiters do, but it's pretty annoying.

> Most of my jobs and all of my contracts have come through referrals by friends and colleagues, and I try to keep them in mind whenever I hear of an opportunity any of them might be interested in. On the other hand,

> there are those who only seem to remember me when they want a contact name or even a personal introduction (a lot of salespeople fall in this category). Those guys are on my anti-networking list.

Networking is an economy - you get out of it what you put in. If you contribute answers and useful information to discussion lists, then you will more likely be rewarded with assistance when you need help (and more likely to be treated with leniency if you ask a dumb question). If you help people get jobs, then you will more likely be remembered when those same people hire or make referrals. Those who only reach out when they need a job, a loan, or introduction to business contacts are as welcome as the pal who calls only when he needs help moving his sofa.

Most importantly, be good at your work. People will remember, and the jobs will come to you.

Don't Waste Time on Recruiters

I've known people to get jobs through third-party recruiters, and I've managed to get some decent interviews and even a couple of jobs (but not good ones) through recruiters. So I'm not saying don't use recruiters, but don't let them waste your time.

> The last recruiter I wasted time on was an outfit called Remington International, a high-flying outfit located in a fancy Westwood, LA office tower. At least one of their staff was honest enough to tell me that if I didn't check with them every two weeks they would forget about me. This was after waiting in the lobby after being warned the previous day not to show up late, and then sitting through a "practice interview" in which I was supposed to give them leads at every company I'd ever worked at. But it was entertaining to watch the recruiters running around giving each other high-fives.

Don't be misled into thinking that recruiters work for you.

> At the height of the dot com boom, I applied for a position advertising a high salary. But the recruiter from the imaginatively named firm General Employment not only assumed I was at his beck and call ("Hey, I scheduled an interview in an hour - can you show up?"), he spent most of his efforts knocking down my asking salary. "It's not 140k, that was a mistake. It's 120. So you'll take 110?" Maybe his rent check was due.

Third-party recruiters work for themselves - they get paid when they close a deal, like realtors (but without the license or two-week course).

Be Prepared

Do Your Research

In this great Internet day and age, it's easy to do some research on your prospective employer. Any self-respecting company, including startups, will have a web site, where you can check out the company history, management and products. Not only will this allow you to form an initial impression of the company, this will give you a foundation for asking some questions during your interview, either to impress your interviewers with your preparedness or to figure out if these guys are for real.

Quiz Show

I don't like interview quizzes (made fashionable years ago by Microsoft and more recently by Google. But quizzes are a fact of life and you should prepared for them. I prefer asking candidates about their projects, why they made the choices they did, and what tools they used. Anyone who actually accomplished anything should be able to answer those questions.

> During the height of the dot-com era, I interviewed a Java programmer who cited involvement in several Java web projects but could not name a single Java classes she used in any of those projects. And yet she expected close to a six-figure salary!

It's easy to forget the various tools, languages and API's that one encounters in a programming career, but take the time to review them and jog your memory before the interview.

Don't Ask, Don't Tell

Important as it is to refrain from discussing race, sex and religion in the workplace, it is even more crucial to keep your opinions to yourself in the interview. There are plenty of federal regulations intended to prevent those factors from keeping you out of a job, so take advantage of that protection. If it means that much to you, you can inflict your opinions, peccadilloes and dogma on your coworkers-to-be when they're stuck with you.

Put Your Best Face Forward

This is stating the obvious, but make a good impression.

> I interviewed a technical writer who was looking to make a move from Chicago to the bay area during the dot com boom. On the cover of the technical report he submitted as as a writing sample, "technical" was misspelled.

Cross your t's and dot your i's.

Don't Look Like a Mercenary

Of course, money is important (unless you're fortunate enough to be young, rich and stupid). But when you're applying for a job, you should at least pretend that money is not your only motivation.

> A coworker introduced me over a cheap dinner to a friend of his who expressed some interest in working for our company. His only real question to me was "how much does the job pay?". Considering he didn't have much to say about himself, and we weren't a large company looking for a Level 2 Software Engineer with a specific pay grade, I had no idea and little interest in coming up with a figure. I couldn't tell how good he was,

> but I was under the definite impression that his primary interest was in finding a higher salary.

Don't Be Late

First impressions make a difference, and showing up late to an interview is just about the first possible bad impression you can make.

> I used to have a terrible punctuality problem, but the worst was when I showed up a half hour late for an interview with iRobot - I left work at what I thought was the last possible moment and then crawled through commuter traffic all along the Charles River. For some reason, I seem to be more punctual these days, possibly because I no longer deal with Boston traffic, but also I try to get there a bit earlier, at least to scope out the lay of the land.

Scheduling yourself to arrive fifteen minutes to half an hour early gives you some safety margin - and if you do get there early, it gives you time to check out the office building and check out the surrounding area.

Don't Be Weird

If you're rich and weird, you're eccentric. If you're interviewing, keep a lid on it.

> I conducted one of the most painful interviews of my life with a guy who started off by professing his infatuation with Asian women ("I love Asian women", I believe were his exact words). I can only imagine he got started on that thread because the previous person who interviewed him was an Asian woman (and in management, to boot). It turns out that was the most interesting thing he had to say for the next very long hour.

Look for Flaws

A new job is an investment - you are committing time in the expectation of gaining income, expertise and connections that will advance you in terms of career, finance and overall happiness. There is an opportunity cost in taking a new job - that is time you could spend elsewhere, learning other skills, completing other projects and meeting other people. So do your "due diligence".

The First Date

Job interviews are like first dates. Everyone is on their best behavior, so any visible flaw you see will certainly be present several times over if you actually "hook up".

> I spent one interview listening to the general manager yelling into the phone at his subordinates. He assured me that he never did that with his engineers (not true) - nevertheless, I ended up buying an iPod to tune him out, and my final meeting at that company lasted three hours, much of it a high-volume blame-fest. A company president with a similarly self-

> absorbed personality complained to be numerous times before I joined about the rudeness and insensitivity of programmers (at least we appreciate irony). That got pretty old after a while and again, when times got tough, the tough got accusatory.

If you see any disturbing behavior, imagine what it's going to be like on the job, especially during stressful times.

Meet the Family

It's not just the boss you have to worry about. Check out your prospective coworkers, and see if they're hiding anyone in the closet. And get a feel for the company culture.

> When I was working on a poorly managed defense project in suburban Maryland, insult was added to injury as I worked overtime with the flu while my coworkers attempted to engage me in "debates" on the evils of gay pedophiles (as opposed to straight ones), interracial marriage, how America was a Christian nation, and what Rush Limbaugh said last night.

A company culture might not just offend you - it can change you, for better or worse.

> At my first job in Silicon Valley, I learned a thing or two about political infighting, but I felt dirty (and physically a little ill), and the moments of wicked satisfaction didn't make up for lost time. Later, I joined a game company filled with so much childish behavior (crying, foot stomping, throwing things - and that was the management) that I wanted to take my toys and go home.

Order the Lobster

One traditional date test is how much the guy is willing to spend. Hopefully, that is no longer the case in this modern day and age. But still, it's something to watch out for when "dating" a prospective employer - a company that is cheap during the interview will be even more stingy with you as an employee, and particularly for small companies, it may be a sign of low operating funds.

> Even companies that are not particularly generous will put on a good show during the interview, but I did have an interview with a company in New Jersey that didn't even offer to reimburse me for taking a train to the interview from Boston (and back the same day). Combine that with their reluctance to give a specific offer when they expressed interest in hiring me, I opted for a job that seemed less interesting but seemed to have more abundant and secure benefits.

> Another employer kept asking me during my interview to think about the least amount of money I needed for my standard of living. In retrospect, I should have responded by asking them to consider the maximum amount of money they could spend on me. And then perhaps I wouldn't have

ended up getting the job, which involved a pay cut, reduced benefits, and ended up with me purchasing hardware for the project so I wouldn't have to listen to the boss make a big deal about spending the money.

Read Between the Lines

Some ominous phrases:

"We're a family." The dysfunctional kind, like the ones in the Jerry Springer Show.

"We have a good core team." Everyone else left.

"We won't deliver a product until it's ready." We're late.

"I know we can do this." All evidence to the contrary.

"We're in stealth mode." We have no marketing.

"I'm really direct." I'm really obnoxious.

"On occasion we need everyone to work extra hours." We have a stupid schedule.

"We have some high-power management ready to come in at a later date." After you do all all the work, you're not getting promoted and someone else will take all the credit.

"You're recommended by X? Then this interview is just a formality." This place is really political.

"Don't worry, I'll take care of you." Promises, promises.

In my experience, your best shot at avoiding politics (if that's what you really want) is in startup companies that have less than twenty people. Once they grow to a certain size, then people start worrying more about their titles and turf then in getting something done and keeping the company alive. But if you're more interested in structure and security, that might not be the right situation for you.

Believe the Rumors

The rumor mill is right. That's not to say you should believe everything anyone says about a prospective employer. But the reputation of a company is propagated by numerous employees, customers, vendors and partners over the years.

So if the word on the street is to turn around and run the other direction, then do it.

Check the Benefits

Don't take any benefits for granted. I started my career at large corporations with standard benefits and relocation packages, so I've been caught by surprise a few times when dealing with small and even some medium-sized companies.

Relocation

> When I got an offer from an interesting startup in California, I assumed they would reimburse me for my move from Boston, but fortunately I casually mentioned the issue just before I verbally accepted the offer. My manager-to-be was surprised by that assumption, but was quite reasonable and came back to me with a modification of the offer - an amount that didn't quite cover the cost of relocation, but was still better than nothing.
>
> With a much larger company numbering several thousand employees at the height of the dot-com boom, I again assumed that relocation was a standard part of the offer, but only after I formally accepted and called up the HR department did I learn that it wasn't. After starting work, a coworker who just started said he had been relocated by the company, so I should have just made sure of this during the offer negotiation.

Vacation

When I started working, it seemed that three weeks of vacation was standard, and I even had one job that provided six weeks a year. But like all other benefits, you can't take it for granted, anymore.

> The first console game developer I joined lowballed me on the salary, which distracted me from noticing that they only offered five days of vacation. And they were pretty stingy with those - even after working every weekend, they would deduct a half day if I was out for an afternoon (and yet still worked the evening)

Some advice I've received but have yet to apply - when you've finished negotiating your salary, also negotiate your vacation. Particularly if you've had a long work history - there's no reason you should start with an entry level amount of vacation.

Health

In the United States, it's a lot easier to lose health coverage than gain it, so this is one area where you should pay special attention. First of all, check that health benefits are indeed available, as technically speaking, companies are not required to provide it. And those that do, may not not activate it immediately or may have certain restrictions.

> For example, worked at one company that had just spun off from a major consumer electronics firm and still carried the same benefit packages - however, the health insurance did not start until one month after the start date. I believe this is fairly common in the entertainment industry.

Make sure your previous coverage will last up to the beginning of your new coverage. Short-term policies are convenient for this situation, but don't waive or terminate your COBRA rights until you've established new insurance. If your new job falls through before your new coverage starts, you don't want to compound that problem by not having health coverage.

Know Your Worth

I remember in particular one piece of advice at Texas Instruments, my first job right out of college - "You would be a fool not to keep track of your market value." And this was from a distinguished computer engineer who'd been at that company for decades.

Industry salary surveys are a starting point, although the methodology is usually sketchy.

> For example, I saw in one issue of the popular Game Developer magazine salary survey that none of the programmer salaries in the survey exceeded $200,000. Then if you read their explanation of the survey methodology, they note that they discarded all reported salaries over $200,000. Which means what? Nothing.

And I suspect that survey results are skewed high - if you feel good about your salary you're more likely to respond to a survey than if you're low-paid or unemployed.

You may be willing to trade off salary for other aspects of a job, but keep in mind, when you take a pay cut, it may not be easy to make up that difference later. Employers tend to negotiate based on your most recent salary, not your highest previous salary. And while employers are quite willing to point out that economic times are slow, the job market is not good, and the cost of living in their area may be lower than others, they are not quite so enthusiastic about offering more when times are hot and they are based in an expensive area.

> The one time I took a significant pay cut for a new job, I regretted it. The decrease was even more significant if you consider the portion that was actually a signing bonus and returnable if I left the company for any reason during the first year, and if you count the markedly lower benefits. The company shares mentioned in the offer letter were conveniently forgotten by the employer, and the job turned out to be quite unpleasant. It didn't take me long to feel stupid - right after I joined, my new boss rattled on about how she was willing to pay twice as much for others she was trying to recruit.

So if you're taking a pay cut, be sure about what you're getting in return.

And (this is old news now since the dot com bust) don't do it just for the stock options.

> When I left PRI, I didn't bother to exercise the startup's pre-IPO shares, reasoning that the money was better spent on a Dreamcast (and it was). When I left Neomar, I did exercise my options, but largely out of politeness - the stock certificate still sits on my wall today.

I will note, however, that employee stock purchase plans in a public company are a good deal. If always sell as soon as possible, you'll make a little something.

Have Contract, Will Travel

"Every contractor working on the Death Star knew what he was getting into" - *Clerks*

Know What You're Getting Into

Back in the early nineties, I had dinner in Boston with an operating systems engineer who worked on a contract six months a year and spent the other half of the year at his cabin in Maine. I thought that sounded like a pretty good lifestyle, and these days I can tell people are similarly intrigued when they ask me about my consulting business. It sounds appealing - be your own boss, make your own hours, pick your own projects, and get paid more than you made as a salary. But don't be misled by the fancy-sounding title "consultant". Sometimes it means just working on a contract basis, and sometimes it's a euphemism for being in-between jobs.

In fact, as a consultant, I earn less on average than I did from a salary, there are long gaps between paying projects (when I first began telling people I was an independent consultant, I felt like I was lying), and some years I qualify for the low-income discount from the utility companies. But I have time for my own projects between contracts, I find the nitty-gritty of running my own business interesting and educational, and (perhaps this is my imagination) I receive more courteous and professional treatment as a contractor and consultant than I did as an employee.

So if this lifestyle still sounds appealing, read on...

Get Ready

Consulting can be very unsteady work, so you shouldn't rely on it as a living if you don't have a good cash reserve. In general, I've heard that you should have six months of living expenses saved up in case of unemployment. For a consultant, I think you should have at least a year's reserve.

> When I first started relying entirely on consulting work for income, I figured that I had six months of living expenses in reserve. Six months later, I realized I was right on schedule.

If you had employer-sponsored health insurance, be sure to continue that coverage under COBRA until it expires or you have found other insurance. Under federal law, you're entitled to health insurance as long as you've taken full advantage of COBRA, but if you neglect to sign up or you discontinue the coverage early, you could be out of luck - insurers are not then required to grant you a new policy. Companies with real HR departments usually know what they're doing, but some small companies play fast and loose with the rules, so don't can't assume they'll do the right thing.

> I've been in a couple of startups that didn't seem aware of their health care obligations - one completely ignored my queries about signing up for

> COBRA, and after the decision period expired I received a letter from the insurer stating that I was no longer eligible. At the other, the owner expressed surprise when I pointed out that California extends COBRA rules to small companies via CalCOBRA, and then gave me a waiver to sign ignoring the allowable decision period.

It's unfortunate that the U.S. health care system relies on employers to provide affordable health insurance, but given that, it's important for you to understand the laws that are there for your protection and to take advantage of them.

Get Paid

Always get paid. With money. You may get offered equity, royalties or the reward of an experience, but take those in account only after discussing cash compensation.

> I was offered a tiny equity stake and no cash to consult for a startup. A year later, the founders asked everyone for the stock certificates back so they could dissolve the company.

If the client is certain about making money off the project, he should put his money where his mouth is and pay you now. And if the prospects are more dubious, he should definitely pay you. Either way, there's no reason you should take the risk.

In some cases, a client will try to renegotiate terms after the fact. In the worst cases, like an old apartment roommate of mine who just assumed I would cover his rent, I've had a few clients who expected me to continue working on their projects even when they stopped paying me.

> A game publisher that hired me to evaluate builds from one of their developers terminated the contract, then a month later sent me a new build to evaluate. When I asked if this meant they wanted to restart the contract, the reply was: "Oh, we thought you could just look at it." But at least they paid. When I informed another client that our contract period was up, they said, "now this is the transition period", expecting me to supply free support while they dithered around with the contract renewal.
>
> The worst transgressor in my experience was a game developer who hired me to help finish the "first-playable" build of a game and told me that that it would absolutely be delivered in five weeks or "there wouldn't be a company anymore." After six weeks, when I asked if they wanted me to continue, the reply was "it's entirely up to you." Apparently there was an implicit "but we won't pay", since the invoice I submitted a few weeks later was met with the classic denial-blame-bargaining phases of weaseling out of a bill. Ironically, that developer had a history of threatening and suing any publishers who tried the same trick on them.

You shouldn't have to do this, but be crystal clear that you expect to be paid. Avoid deadbeats. Check your network - few companies have pristine reputations, but some are repeat offenders, to the point where they effectively have bad credit and contractors won't deal with them. Don't take "...because of our financial situation..." for an excuse. (I did on one occasion just so I wouldn't have to listen to any more whining, but righteousness is expensive) If they can't pay their debts, they should declare bankruptcy. Any client that wants to be treated as a charity should reincorporate as one.

Wait 'til the Check Clears

As a teenager in Iowa, a bus driver noticed I'd tried to get on with a few cents short of full fare and gave me a succinct lecture: "if you don't have enough, just tell me." I wish he could set some of my clients straight. In my experience, most clients do pay up, but when they're having trouble coming up with the money, they don't tell me what's really going on.

> For eight months I had no problem submitting invoices to a particular client until one day the accounts guy who'd processed of all my previous invoices asked "Do you have authorization for this?". The next time I tried to submit an invoice, that guy was gone and another person directed me to drop the invoice off in a nearby office - an empty office. It was no surprise when I inquired about it six weeks later that they had no record of it.
>
> But that was better than the company whose office flunky took my invoice and then went to Taiwan for a long vacation. Or the company who told me they couldn't complete the check because "the girl who does that is manning the switchboard right now". I mean, really.

When you start getting "the dog ate my homework" excuse, it's time to wrap it up. And don't spend any of that money until the check clears.

Don't Sell Yourself Short

As a rule, a contract rate should be higher than a salaried rate. Prospective clients that act surprised at that are either disingenuous or not too bright. Remember, the client isn't providing you employee benefits such as health insurance and training, paying your employment tax, or paying you severance when the contract is terminated (unless you have negotiated it). You have to cover your own insurance costs, pay a fifteen percent self-employment tax in addition to regular income tax, keep yourself up-to-date and marketable, oftentimes provide your own equipment and facilities, and cover your living expenses in the periods between contracts.

Availability of contracts is largely a matter of timing. When business is good, that means you'll have to turn down some projects while you're occupied with others. Taking a low-paying contract may mean you'll passing up a higher-paying contract in the meantime - you're paying an "opportunity cost".

> During just one year, I missed out on at least three potential contracts because I was busy on others. In one case, I passed on a one-month full-rate job that would have paid twice more in total than the lower-rate two-month project that had my attention at the time. There were some other

> considerations on the lower-paying project that hopefully made it worthwhile, but it's hard not to feel regret when you've traded off that much income.

Longer-term contracts don't necessarily justify lower rates. A contractor friend of mine told me when I started out that companies can terminate a contract at any time, and you just have to factor that into your price. And indeed I've found that to be true - most clients will have no compunction about terminating a contract if the funding falls through, or if they think they can get someone cheaper to do it.

> One client pouted that it wouldn't make sense for her if I didn't stay on project for the duration, expected to be another year or so. The entire project was cancelled a few weeks later, and I didn't even get paid for all of that. I have to say, that didn't make sense for me.

It's unfortunate that some clients will stoop to haggling like car salesmen ("I'm writing down a number....").

> I had one client who not only bargained down my rate ("that's more than I make!", he exclaimed), but also wanted payments tied to milestones, ostensibly as as formality. And every couple of weeks, they had just one more favor to ask, until, by the second contract renewal, they were cramming as many listed but unspecified deliverables as they could into the schedule on a weekly basis without even making payments for each deliverable, and trying to sneak in unpaid "transition" periods of work between contract renewals.

If a client acts cheap before the contract even starts, it's just going to get worse. Phrases to watch out for: "We're on a budget", "This should be easy", and, my favorite: "I would do it myself, but I don't have time." Avoid these projects - it's not worth the aggravation.

Make Your Reputation

Every project you get is a potential link to your next one. One executive-level consultant told me he used to try marketing himself aggressively, handing out business cards at networking parties, etc, but all of his work came out of his existing network. This has been true in my case, too - all of my contract work (and many of my salaried jobs) have come through referrals.

So it is important to maintain and enhance your reputation on each project. This includes getting quality work done, and managing expectations.

> I've heard complaints from clients about having to throw away work performed by previous contractors just because no one could understand their code. And I've heard of contractors who will actually offer one rate to do a quick and dirty job, and a higher rate to do clean code with good documentation. That shouldn't be a choice - do the job right and be

> honest about how much it will cost and how long it will take, even if it's not what the client wants to hear.

Besides good work, you also want a reputation for billing fairly and not trying to milk each project. I've seen contractors bill the maximum allowable hours no matter how much they actually work, and contractors who try to extract every little perk and expense reimbursement, down to free sodas and per diems. This makes contract negotiation harder for everyone. Forget the small stuff - it's the big payment that counts, and if you have a reputation for billing fairly, it's easier to maintain a higher rate.

And most of all, don't make the person who referred you regret the decision.

> One programmer I introduced to a game developer embarrassed me by being overly inquisitive and pushing aggressively for a demo. Another artist I introduced to a potential client kept probing me for inside information while negotiating a contract. (and meanwhile the client was doing the same. What a headache) I'll never refer those guys again - too much damn trouble.

If employment is like a marriage (or at least a steady relationship), then contract work is like prostitution. And I don't mean that in a bad way.

> I worked with one contractor who emailed the whole company calling for everyone to strike in support of better conditions for the office cleaning people. Well-intentioned, but I doubt even the cleaning staff was going to rally around him.

The most important thing in the contract relationship is that you get paid. As a mercenary, you can offer friendly suggestions about how your clients run their businesses, but it's their business.

Put the "Contract" in "Contractor"

My least favorite part of contract work is the contract itself. The conventional wisdom is "Get it in writing". If someone is going to cheat you, they're not going to let a piece of paper stop them, but it doesn't hurt to have something that you can fall back on in the event of legal recourse, and even for "handshake" agreements, people can have different memories of even the basic terms, so you want to get that written down in some form.

Typically, the client will present you with a contract put together by the client's attorneys, who are solely representing the client's interests. It's up to you to protect your interests. So don't get bullied into accepting any terms that make you uncomfortable. Statements you should ignore:

> "This contract is intended to protect your interests." Bah! The company's attorneys are bound to represent their clients' interests, and that does not include you. Go ahead and ask the company's lawyers if they are representing your interests.

> "No one has complained about this before." That's probably not true, and if it is, it means they didn't read it carefully or bother to have their own lawyer look at it.
>
> "I have to protect my company's interests." So what? You have to protect yours. Lifelong servitude and your first-born child would be in your client's interests, but that doesn't make it reasonable.

Sometimes it's hard to get a clause removed just by asking, so one tactic I've taken is to make sure the clause goes both ways. After all, the contract is between two business entities, so it should be fairly symmetric.

> When faced with a non-poaching clause that stated if I "encouraged" any of their employees or contractors to leave the company within a year of the contract's duration, the penalty would be $100,000 or a year's salary of the employee, I asked that the clause apply both ways (hey, I could use $100k!). The clause was conspicuously absent in the next draft of the contract.

Just because the contract was put together by an attorney doesn't mean that it's worth the paper it's printed on. As with doctors and automobile mechanics, it's easy to assume lawyers know what they're talking about. But sometimes they do, and sometimes they don't.

> An in-house counsel for VC-funded startup assured me that the indemnification clause in my contract had nothing to do with financial liability. When I insisted that "indemnify" meant "to pay", she looked it up in her pocket dictionary and had to agree. But I guess she hadn't authored that clause - I later found a word-for-word copy of the contract on a sample-contract web site.

Another piece of conventional wisdom is to have your own attorney look at your contract. But attorneys are expensive. In general, I don't bother with an attorney if the contract looks straightforward, especially if the contract is small. If there's anything that is not clear to you, you should definitely have an attorney look it over (or walk away if the contract pay is too small to warrant the contract hassle and legal fees). But as before, keep in mind that not all attorneys are created equal:

> An attorney I picked out of the yellow pages (I was in a hurry) to review a game development contract gave me pause when he referred to "newfangled" video games. He did provide some useful information and corrected misspellings in the contract (I am constantly amazed how someone charging $300 an hour to put together a contract doesn't bother to run a spell-checker on it). But I would have felt more secure with someone that had knowledge of the industry.

If you get an attorney, find one who is capable and knowledgable in your industry. And pass the name on to me!

Keep Busy

In between projects, you may be inclined to panic. If you do, you're probably not cut out for this business. Downtime is an opportunity to catch up on things that may be neglected while you're busy on paying work.

You can keep yourself up to date by attending technical presentations, conferences and trade shows. It's an opportunity to see the latest products and technologies, and perhaps investigate new areas in which you can work.

> I like to go out of town for an industry show after wrapping up a big project - usually the more stressful the project, the farther I like to travel. It's a nice opportunity to unwind, get away from the computer for a few days, and see what I've been missing the last several months. I usually look for shows with free expo passes rather than pay for a full conference registration (not for any good reason besides being cheap). Watch out for spam, though. I've been unsubscribing from CMP missives for years, now.

You can participate in technical discussion groups. It's an opportunity to toss around ideas, identify people you might want to work with (particularly in local area groups), and help out people who might help you someday.

> I've kept up with the many interesting developments in Java and Java-based projects by following the local (Orange County and Los Angeles) Java User Group mail lists. I don't particularly like sitting in meetings, so I eschew the regularly-scheduled presentations, but the discussions are often illuminating, and they keep me informed of the latest promising tools. I even brought in one group member as a co-contractor on a project.

You can contribute to open source projects. It's an opportunity to communicate with some excellent developers around the world, research open source tools that you may find useful, and give back to the community.

> During my first time off, I made some fixes to JFOR, an XSL/FO-to-RTF converter, so I could use it to generate Word-readable versions of my resume from an XML Format. Since then, I've decided recruiters and HR people should know how to read and cut-and-paste from PDF, so I don't use JFOR, anymore, but I'm gratified to see once in a while an acknowledgment of my contributions from the JFOR list.

With all of these activities, the idea is to improve your capabilities and marketability, expand your network and establish some street cred, and enjoy yourself in the meantime!

So You Want to Make a Game

"Once you can remove no more, you are done." - Japanese saying

Getting Started

My first piece of advice to anyone contemplating a game development project - don't. (Well, at least not until you know what you're getting into.) But if you go ahead, anyway, here are some tips beginning with staffing:

Dispassionate Gamers

Game job postings typically list "a passion for gaming" as a prerequisite (along with its partner, "willing to put in extra hours"). Other industries don't do this - you don't see ads requiring "a passion for factory automation software" or "a passion for working on banking systems". Anyway, a devotion to games doesn't necessarily translate to productivity and work ethic.

> Most of the game developers I've worked with were avid gamers but many, including one of the best gamers I've ever met, didn't apply the same concentration and motivation to their work. I've seen plenty of work with obvious defects, even an FMV with a two-second blank gap, either due to inattention or laziness (one artist even told me his video package could reexport his FMV in AVI format, until I stated that was the only format I could process for our target console - then he promptly reexported it).

I'll take professionalism over passion any day. If you need to hire janitorial staff, you're not going to look for some crazy nut who has a passion for toilets - you want responsible people who take pride in doing their job.

> Most of the best game artists I've worked with were only moderately interested in gaming - they were artists first and their results far surpassed those of mediocre artists with high gaming skills. They were conscientious enough to play-test their work during development but once the games were released, typically never touched them again.

Now, game design is one area where presumably an absorption in games would be really useful. But even there, you'll find the best game designers are who have the intellectual breadth and analytical ability to figure out what makes a game fun and to move beyond knockoffs of existing games.

> One designer who was an extremely good skateboarder and skateboard game player decided to differentiate his game from all the Tony Hawk clones by removing the fun parts, resulting in an obvious knockoff that was unplayable.

A common entry route to game development is via QA groups, and I think that is a good one - testing is a good introduction to the game development process without getting in the critical path, and anyone who can't develop the patience and analytical abilities for that role is probably going to do a worse job in a production position.

To put a final point on this, if a passion for gaming is so important for game development, why are so many non-gamers in high-level decision-making positions on game projects? Many, if not most, of the high-level producers I've met are mediocre gamers at best, and it gets worse as you go up the corporate ladder.

> I know of one game prototype that had all kinds of pizzazz thrown in at the producer's request, and then the whole thing had to be scaled down at the last minute so the demo wouldn't confuse the executives who had final approval. (Of course, the producers should have known this)

Programmers are from Mars, Artists are from Venus

The production pipeline is always the bottleneck, so you need people who not only work well, but work well in the context of a production.

Even some experienced and talented game artists do not work well in game development teams. Just throwing assets over the wall and assuming the programmers will fix it wastes time and engenders hostility from the programming team. The more difficult artists I've met were stubborn to the point of belligerence and considered the rest of the game team as staff supporting (or limiting) their artistic accomplishments.

> Some phrases that have lost their novelty - "I didn't change anything", "It's broken - fix it!", "That's how it works in 3D Studio Max".

And while programmers often grouse with justification that artists aren't feeding suitable data into the production pipeline, it is incumbent upon them to provide the proper guidance, in the form of clear documentation and explanation.

> Programmers often don't like to commit to numbers early, but artists and game designers need reliable asset budgets to do their job within the proper constraints, just as programmers need to know the hardware and delivery constraints. If artists and designers actually ask for budgets and guidance on how to optimize assets, then by all means accommodate them.

The Producers

Producers can be useful, if they're not puffed up by the term "producer". They're the only the part of the production pipeline that doesn't actually produce anything (except those producers who produce funding - they get to be called "executive producers"). A better term would be "facilitator".

In the worst case, a producer will just cause more work for everyone else. The last thing a game project needs is a high-maintenance producer.

> I was once plagued by a throng of producers who would hold daily status meetings, carry around lists of tasks and ask everyone to update the number of hours remaining per task on the lists (none of which matched), and stroll around asking, "So...what are you working on?" After may of the producers were laid off due to tightened funding, the project ran more smoothly.

But a knowledgable producer who is focused on keeping things running smoothly is indispensable.

> I was vastly relieved that a survivor of the aforementioned layoff was an assistant producer who did a fantastic job of keeping on top of things. Every time I needed clarification on a game feature, he had the answer, knew where to get it, or could make a reasonable call on the spot. In any case, I had an answer within minutes. And he was the guy who ordered all the late-night meals.

At least producers in development shops have an idea what it actually takes to make a game. Producers who've only worked for publishers are less knowledgable and more self-important.

> One stupid publisher trick: planting a producer at a developer's office during crunch time to make sure they're all working late into the night. When I showed one of these producers, who was camped out in my office during one of these crunches, the set of game design, production and business books on my shelf, he commented his boss had recommended he read some of them. And then he kicked me out of my office so he could have it to himself.

Think Inside the Box

"Think outside the box" is an oft-misused mantra. Questioning your underlying assumptions brings about innovation. Just making stuff up is a waste of time.

> In high school, I joined other students who wanted to pad their college applications by competing in a statewide brainstorming competition, where we attempted to outdo each other in constructing the most fantastic scenarios possible. Odds favored those who most lost touch with reality.

What separates game artists and programmers from their brethren in other fields is the ability to create for resource-constrained platforms.

Screen Size

Take into account the screen size. For PC games, you have to decide how many standard PC monitor resolutions, refresh rates, color depths, and full-screen vs. windowed modes you're going to support, and be sure to test the game with those settings.

Even with console games, you may have more than one setting to worry about. If your game will be in both the US and Europe, then you need to handle both NTSC and PAL, which have different screen resolutions, with corresponding aspect ratios and memory requirements, and different refresh rates, which may affect any game behavior dependent on per-frame computation. And there are other modes like EURGB60, M-PAL, 480p (progressive scan) and multiple levels of HDTV.

Preprocess Everything

Data created by game artists and designers eventually gets converted into formats usable on the target platforms. PC game engines often defer this conversion until runtime for convenience, but for consoles, where memory is comparatively limited, loading from disc is slower, and the main CPU may be relatively underpowered, it is important to have as much data preparation and optimization done offline as possible. Even for PC games, while developers may be lulled by the latest and greatest in PC hardware, there is still a customer base with configurations a few years old, and if they were budget PC's then, imagine how limited they are now.

Get Lots of Hardware

Game development schedules are always tight, and even expensive hardware is cheap compared to personnel cost and the cost of missing a milestone and having your project cancelled, or missing the holiday retail season and losing out on those sales. Cutting-edge hardware, especially console development kits, is notoriously fragile, so you want to have extra units on hand if and when the hardware fails.

> Almost every time I've worked on a console game, the game development hardware malfunctioned at some point and had to be sent back to the console maker for repair. In each case, the turnaround time was no more than two weeks, but two weeks on a crunch time project with monthly milestone deliverables is crucial. Fortunately, the developers always had on hand an extra kit that could be repurposed from less critical tasks.

Another reason to have redundant development hardware is to identify bugs that are due to glitchy hardware versus those present in your game.

> Near the end of one console project, I ran into crashes of our game that occurred after several hours of the game running idle. Since we had multiple test kits, I could run several soak tests in parallel and isolate the crashes to one unit, and thereby conclude that it was a test hardware problem (and it was verified later that some units were known to have overheating issues)

Console games have the advantage that you only need to verify proper operation on a very limited set of configurations. For PC games you should have a variety of different hardware, operating systems, and various configurations for testing your game. This is true for cell phone games, too - phone models vary in screen size and color resolution, refresh rate, memory, etc.

> I nearly missed one Windows compatibility issue with a PC game after Microsoft introduced a service pack that removed support for a video

> codec that we used for an FMV (in fact, the codec was used by one of our middleware vendors). None of the development or test machines used by the developer had this service pack, but fortunately the publisher's salespeople noticed the problem when they installed the game on their demo laptops. (when you're relying on your publisher's sales team for QA, you're just asking for it!)

Less is More

More than in most other software fields, game development is about efficient deployment of assets.

Think Small

Game designers and artists often believe it's easier to create more content than you need and pare it down as needed than to start small and add. This may be true in the "micro" sense but poses huge risks in the "macro" sense, particularly for console game development. One of the common "crunch time" factors in console games is the late attempt to get the game running in console memory. Usually this problem is hidden until an inconvenient time by the fact that content developers usually develop on PC's (and sometimes XBox's) with high-performance graphics hardware and several times more memory than a console. And programmers, too, will develop on console devkits that feature more memory than the retail consoles.

> Many designers and artists complain about the constraints of video game development, but I was gratified to encounter one exception. A junior game designer who was ordered to pare down his level commented to me that his level was actually improved by the streamlining - it forced him to make sure everything that he did retain was effectively used.

Constraints are a good thing - they keep us from wandering all over the place trying out everything possible and instead focus on validating the cliche - quality over quantity.

Every Polygon Has a Price

Each polygon, texture, and frame of animation should be justifiable. Serving as "eye candy" isn't enough reason to put something in.

> On one front end I spent quite a bit of time debugging some animated hieroglyphic textures, only to find out later that those icons had no connection to the game at all - they were just there for artistic, but meaningless, effect!

HUD's in particular tend to echo the worst of web design, ranging from the early blinking text to the modern Flash-filled pages. Games should be no exception to the principle that the interface should not get in the way - the best interfaces are interfaces that you don't even notice.

Sound Advice

What goes for graphics, goes for sound.

> Laboring under the misconception that more options are better, one game developer president stated that our extreme sports game should allow ambient sounds and the soundtrack to be played concurrently. It turned out that loud rock songs easily obscured twittering birds in the background and hardly warranted the extra development complexity (the console had hardware support for just one stream). Another game had a list of sound effects for every element on the HUD, potentially resulting in a Las Vegas slot machine effect - the sound designer threw out most of them.

More is not necessarily better, and often it's worse. Imagine all the sounds that could go off at once doing so, and scale it back if the result is cacophony.

Dialogue

Same goes for dialog - every line should have a purpose. We don't want voice-over (or text-over) dialog distracting from the interactive flow of the game, and each piece of dialog requires space, scripting, services of a voice actor (if voice-overs are used), rework if the dialog has to be rewritten or different voice actor is selected, and translations and re-recordings if the game is localized for different regions.

> On one game project where I script-doctored the dialog, the publisher looked over the results briefly and asked for some crowd NPC dialog in one of the "cinematic" fight scenes. I added it just to keep them happy, but sure enough, once the level designers worked in the voice-overs, it was just a big muddle.

One practice I have used in writing game dialog is to include notes at the beginning of each section explaining what the dialogue is supposed to accomplish. Some of these objectives are the same as in any story, e.g. increasing empathy for the player character, establishing the badness of the bad guy. But dialogue can also highlight aspects of your game - if you hear the NPC's talk about what they see or hear and how they coordinate amongst themselves, then you know the capabilities of the game AI.

Another thing to keep in mind is that scriptwriting for games is really more like scriptwriting for animated features rather than film. As with the former, you can't rely on the range and subtlety of visual expression conveyed by human actors, so you must make sure it's clear in the words.

First Things First

As a general rule, anything that can be completed early in the project should be completed early. Get it out of the way, let it get tested thoroughly, and leave crunch time for the hard stuff, of which there will be plenty.

Front End

The front end is typically an afterthought, fleshed out in the final months of a the game development, but it really should be one of the first things implemented. Even if the front-end requires some assets that will not be finalized for a while, placeholders can be substituted.

1. Completing a front end early provides a real functioning component of the game that can be demoed immediately and shows you've got more than just some storyboards and mockups with Flash.
2. Designing the front end early forces the game designers to complete the game design to that extent, so critical decisions like game modes (single-player, multiplayer, story, arcade, etc.), scores, game-save interface, are resolved early.
3. Implementing the front end early gives the developers (and anyone else who sees the game, including the publisher) a specific idea of what the game is about. A common complaint among developers on game projects is that they don't understand what the game is supposed to be about - starting up the game in the same way as the eventual customers can alleviate that problem.
4. Implementing the front end early will expose design flaws, logical inconsistencies, and potential incompatibilities with the console makers' requirements, e.g. memory card usage. Developers usually implement shortcuts that start up whatever level or feature they're working on or testing, but then they get in the habit of relying on these shortcuts and the front end is not well-tested.

Media

The final distribution on media may seem like the last thing to take care of, but again, it's something that can and should be done early. Preparing the game assets for the target media is typically an elaborate process, and it's best to get a handle on that before crunch time. And ideally, you want to test the game running on the media as soon as possible, so you'll know if load times are acceptable, sound and FMV streaming works, and even if the game fits on disc.

Many games now depend on "world" streaming, so it is crucial to verify that disc performance can keep up. Console development systems often provide disc emulators, but the performance characteristics often do not properly match those of the real hardware. I've seen games work perfectly on the emulator and then, a rude disappointment, fail when run from disc, sometimes just before a required milestone delivery.

First Playable, Last Shippable

Programmers complain the design is late and the designers complain they can't finish the design until the programmers have the game up and running so they can tweak it. They're both right. So while it's probably not a bad idea to do as much in preproduction as you can, you can hedge your bets by working toward a "first playable", consisting of at least one, but no more than three levels of the game.

1. This allows you to start off your project with a smaller team while you implement your core technologies and work out the basics of your game, and then you can staff up later to crank out the remaining levels.
2. The smaller target allows you to get to the playable point faster than if you tried to develop the full game at once. This allows you to reach a point much earlier at which you can evaluate the gameplay, asset budgets, target performance, and if the result is satisfactory, then you have a demo for the trade shows and game magazines.
3. If it turns out that the game is fundamentally flawed (or just not what the publisher wants), which is not uncommon, then you can change direction or even start over much more easily than if you had invested a full team and spent much longer in getting the game to a playable point.

Note the first playable is not the same as a prototype, which is basically a minimal demo that you shop around to get the deal. The first playable is really playable, which means it has all the user interface elements, game saves, functionality and polish that you would see in the final game.

Going International

Localization is another typical afterthought. But as with everything else, get things ready early, so you won't have to deal with it in the crunch.

Who's Your PAL?

PAL resolution is slightly different from NTSC - in particular, the aspect ratio is slightly different, so you'll want to verify that 2D elements in particular, such as the front end, text, HUD and movies, still look OK. The higher vertical resolution of PAL also can mean greater main or video memory usage.

> I was astonished to find in one huge game project that none of the programmers had access to a PAL-capable PS2 devkit. They resorted to kludging in the code and then sending off a special build to QA for them to see if it really ran.

The PAL frame rate is also different, so anything in your game dependent that assumes a certain frame period, e.g. movies or code executed per frame that doesn't take in account real time elapsed, will behave differently.

Watch Your Language

I've seen more than one project where it was assumed that the publisher would deliver localized assets (text, audio) just once, and that the assets would be correct and final. That's a pretty unrealistic assumption - publishers will express the importance of getting a game finished on time in no uncertain terms, but when it comes to deliverables from their in-hous departments, don't expect them to respond with the same urgency.

> At one developer working on a console title for the US and Europe, the president of the company assured me that the localized text from the

> publisher would be correct and final, so not to worry that the deliverable would happen just before the ship date. As it turned out, the holiday retail season came and went as we went through several iterations of the localized text, with 3-4 weeks between the updates. One of the near-final deliveries was incomplete because our producer contact took off for Christmas vacation without bothering to give us a complete set of corrected translations (and he didn't mark which ones were changed - that was considerate!)

When you do actually receive translations, chances are they're not going to be suitable.

> The translation may be inappropriate for the desired context. For example, "button" could be translated differently depending on whether it belongs to a controller or piece of clothing. Or the translation may be just plain wrong. I've received translations that left me wondering if the publisher had just hired some beginning language students from the local college. In many cases, I've had to rely on European coworkers and the internet to find the right translations.

Console makers typically require certain phrasing as part of their submission requirements. This can cause your submission to be returned several times until you get it right.

> On one localization project, we went through several iterations of translations for the various mandatory disc and memory card error messages, until the console maker published a set of recommended translations for all of them. Upon which I gladly tossed out all of our publisher-provided translations.

Finally, translations may not fit in your existing on-screen buttons and other user interface elements, so you'll either have to find shorter translations or rearrange your screen. You'll probably need to load multiple new fonts, increasing your asset budget, and if you have a type-in screen, arrange to display all of those characters. All the more reason to set up your game for localization early and avoid rearranging a bunch of your screens later.

IT'S SOFTWARE DEVELOPMENT

Although game projects bear an increasing resemblance to Hollywood projects, game development is still fundamentally software development, yet despite being possibly the most challenging form of software development, the game industry lags behind other industries in software engineering and management practices.

Office Space

It is conventional wisdom by now, supported by studies, that programmers are more productive (and less irritable, I might add) in their own offices. The elite technology companies like DEC used to make a point of getting programmers their own offices (if you were senior enough, you got a window).

> One of my favorite employers, BBN, was one of these companies steeped in technology culture and provided individual offices for each of their engineers, until the fiscal pressures of the early eighties prompted them to move to cubicles and euphemisms ("come see our new open office environment!").

The game industry, on the other hand, is the only software business I know of where people, including engineers, talk about cubicles and open "bullpens" as productivity-enhancing. This is a case where the appearance of activity and interaction doesn't mean more productivity - more likely there's an inverse relationship.

> In one particularly abysmal game company, I was crammed into an office with two other programmers, with the finicky one sitting three feet from me complaining that my typing and mouse-clicking was too loud. (Apparently, he'd never heard of headphones) As a bonus, the men's toilet on that floor was not designed to handle twenty male game developers and clogged up constantly (now I know why the British call it a "bog") - Combined with the lack of air conditioning, it made for swampy weekends.

Not the best environment for the regular workday, much less long work hours, which brings us to....

Avoid the Crunch

Crunch time happens everywhere, but outside the game industry it's recognized as a failure of management. In the game industry, however, even while paying lip service to the notion that crunch time is a bad thing, publishers and developers trot out the same trite spiel - it's an unavoidable consequence of industry pressures, management has to tell developers to stop working and go home because they are motivated to work so much!

That's quite a conceit. It's one thing for someone to get on a roll and work through the night to get something cool done, it's quite another for management to mandate 12-hour work days and weekend shifts for months at a time. It's ultimately a self-defeating proposition.

1. Everyone has a natural pace and level of productivity. The reliable key people on your team will put in the extra hours, anyway. Everyone else will just hang around the office longer, web surf more, and just get on the nerves of those who are really working. In either case, they will resent the disruption of their personal lives due to mismanagement of the project schedule. And they won't be completely wrong.

2. Contrary to popular belief, crunch time does not engender an increased level of production - you get more code and assets, but you get more mistakes due to rush jobs, weariness, and decreasing motivation. The risk-reward balance of crunch time is becoming more dangerous as game projects get bigger and more money is at stake, especially for console games that are unpatchable and the only recourse for flaws introduced at the last minute is to recall the product from the retail shelves.

The Peanut Gallery

The cool thing about games is that everyone's got an opinion. The bad thing about games is that everyone's got an opinion. Playtesting provides critical validation of you game, but sorting the wheat from the chaff is required.

Focus Groups

The problem with focus groups is a lack of. General comments often support the industry's reputation for catering to adolescents - "Bigger boobs! More visual effects!" As with formulaic big-budget Hollywood movies, these elements may be reliable ingredients for drawing mainstream interest, but do you really need a survey to give you this information? Used in this manner, focus groups are just a pseudo-scientific way of reinforcing current preconceptions.

> A game company president was peeved that most of the team showed little enthusiasm for making our game more risque, so the only comments she emphasized from a focus group test were the juvenile ones asking for more skin (the sole female tester demurred, but her opinions were not considered useful). However, the game was based on a family-friendly license and was supposed to achieve an "E" rating, so we spent a lot of time at the end of the project toning things back down.

But focus group results can be useful, if you really focus on "finding the fun".

> I was particularly impressed with one console maker's rigorous in-house play-testing. Their testers would report initial impressions of the game, their impressions after one hour, then three hours, then eight hours. At each step, they gauged their level of empathy with the player character, their satisfaction with the visuals and audio, their frustration level if any with the difficulty level, and their motivation to continue.

Publishers, Who Needs Them

Unfortunately, you do. At least if you're working on a console game. As in Hollywood, publishers think that because they make their money off games, they know how to make games. And since they make money off one in ten games, they know what consumers want. And then there's reality.

Do not let a publisher design your game. If they were qualified to do that, they could develop the game themselves. Publishers should stick to their knitting - providing licensing, tools, testing, marketing, submission requirements, translations for localization, and most importantly, payment. Remarkably, I've seen publishers fail miserably in all these areas.

> On one project I worked on, the publisher was late with the console development kits, borrowed more equipment from us for their other projects, paid late on almost every milestone, lost the license and then complained the game was too boring to sell (and it was fun with the license?), got briefly excited enough about the marketing campaign to

> order a case of custom-labelled Jones soda, and then lost interest and drank all the bottles.

A commonly-stated excuse is that publishers are "just trying to make money". That in itself is enough reason for developers to make sure they protect their own interests, whether it's making a great game or just surviving. But as with any other business, such a statement is overly simplistic - any publisher you deal with is a mix of people who are out for personal glory, personal wealth, job security, and some may be actually trying to make money for the company and some may actually be trying to make a great game.

> I heard one publisher complain that they helped unknown developers get their start only to be left behind when those developers ended up on well-known franchises. Yet this publisher had a history of imposing tight schedules and low budgets on these hungry developers, and they would sell the resulting titles to other publishers at the first opportunity to make a quick profit.

The most successful game companies control their own fate as much as possible.

> The most successful game I've worked on relied almost entirely on in-house resources. An impressive marketing effort, including a fan site, magazine interviews, developer blogs, and even a comic book. The in-house QA group was knowledgable about console submission requirements.

Don't Be Evil

Publishers aren't just often bad at their business - they are notoriously bad about business.

> In just the past few years I've heard of publishers going on group outings to strip clubs, granting projects to developers in return for trips to strip clubs (sense a theme here?), producers propositioning female developers, and even rumor of a publisher bribed with a new Porsche to get a project. Aside from the sleazy, there's the adversarial. I've heard that a publisher lawyer at a game development conference stated the best negotiated deal for a publisher is one that bankrupts the developer. With this attitude, it's no wonder that a favorite publisher trick to enforce crunch time is to send a producer to camp out at a developer's office and make sure they're working into the evenings), And not only have I seen publishers make lame excuses (or no excuse) for late payment and non-payment to developers, I've encountered those tactics in my own contracts with them - contracts rewritten without informing me, micromanagement in order to penny-pinch ("spend ten minutes on this, fifteen minutes on that..."), and weaselly attempts to get free work ("Oh, we thought you could just take a look at it...")

But as easy as it is to blame industry woes on publishers, developers who engage in the same practices have my utter lack of sympathy.

> The one time I heard management say they were opposed to mandatory crunch time work schedules, they shortly announced six months of required weekend work for the entire staff. And I've seen the same gamut of bad-client practices from developers, ranging from renegotiation of ongoing contracts (one client had a practice of this with her contractors, even telling me at the end of a contract "when I have time, I'll let you know what I think is reasonable"), to blatant non-payment (on a project that was dragging on, the client said they had no expectation of paying me for the extra time, yet sued their publisher for the same thing), to sneaking in as much as possible into a contract (I started out at one developer with four weekly paid milestones which turned into two payments for four milestones which turned into one payment for four completely unspecified milestones and an unpaid "transitional period" of work.)

At times it seems developers and publishers are engaged in an unholy alliance.

> I've heard of developers expressing their "appreciation" to publishers with tokens ranging from gift certificates, birthday gifts, hotel accommodations, strip club outings, and even rumors of a car given to a producer in order to seal a development deal. I saw one breach-of-contract lawsuit filed by a developer against a publisher settled by not only payment of cash to the developer, but also a hefty pile of stock and a "consulting" agreement granted to the developer's president. It's not uncommon to see the owners of a game developer shutter the company, leaving the employees unpaid, only to start up a new venture. Who suffers? The rank and file.

I'm the last one to defend publishers, but sometimes you guys deserve each other. Don't be part of the problem.

Startup Without Falling Down

> "The definition of insanity is doing the same thing over and over and expecting different results" - Benjamin Franklin

Knowledge

Know Thyself

Self-awareness isn't a common trait among self-styled entrepreneurs, but a little introspection (not to be confused with self-absorption) can save you a lot of trouble.

> I've seen drama queens who had to be the center of attention (regaling employees with personal stories for hours at a time) and seemed to get bored if there wasn't a big fight or emergency going on. And I've seen control freaks micromanage all aspects of their businesses (to the point of giving employees pop quizzes and approving individual secretarial office supply purchases). Which is fine - passive, unambitious people need jobs, too, and there's plenty of room for small companies specializing in services and small projects. But micromanagement doesn't scale, talented employees don't like to be treated like children, and a large company has to run like a machine, not a series of bar fights. So when these companies succumbed to grand ambitions of growth and innovation, the result wasn't pretty.

Changing your personality is as likely as changing your shoe size. So pick a company that fits.

Know What You'll Do

The glamorous stories about startups include tales of persistence, passion and nights spent on the office couch. But that isn't always necessary if you just want to be your own boss.

> From experience, I know I can maintain a comfortable lifestyle working on one or two contracts from referrals every year and take it easy between contracts. And I know other self-employed developers who play a lot of golf, spend a lot of quality time with their families, and don't work weekends.

But that's not going to scale into anything big or popular. Be realistic about how much your ambitions will match your own follow-through.

> I know of one outfit that put a customer forum on their web site and didn't monitor it for weeks at a time. After seeing it filled with spam and customer complaints of neglect ("treated like a two-dollar whore" was one phrase), they took it down. Another startup asked me to introduce them to high-level game industry contacts to sell their product, but never got around to that little detail of creating the product.

You should be aware of what you're willing to do. And what you're not willing to do.

Know What You Want

Besides knowing who you are, know what you're really trying to achieve. There's the mission statement you put in your business plan to impress VC's, and then there is your real mission. Whether it be getting rich, getting famous, showing your dad you really amounted to something, or getting even with all those people who did you wrong at your last job, you should know, as they say in acting classes, "what's your motivation?"

> I was recruited by the president of a game development company who told me she wanted to make a "great" game. But she spent more time talking

> about famous and rich personalities in the industry than about notable game designers or learning about game design. After lots of schmoozing and mediocre development with junior designers who had to clock in like Walmart employees, the result was, mediocre. And no one got rich and famous.

If you know what you can do and what you want to do, maybe your method will match your ambition.

Know What You Know

It's also important for the company as a whole to stick with what you know. It's hard enough to set up a company, find good people, and raise money. Having to learn an unfamiliar domain and gain connections and credibility in that domain will make success nearly impossible.

> These days, a lot of people want to be Steve Jobs. I was brought into one successful software company that was taken over by a group who had made their fortunes in sales and services, and in short order they managed to run it into the ground. Status meetings consisted of "when can I sell this?" The delivery process amounted to "you're giving me a build today." A process perhaps fit for a burger joint ("you want fries with that build?"), but not for software development.

A significant successful project requires all kinds of people - and chances are, you're not all kinds of people. If you're a technology guy and not a deal-maker, you need to bring in or partner with people who can bring in the deals. If you're a company builder and investment getter and not a technology guy, you probably shouldn't be architecting a new software product. Figure out the critical functions that should be your focus, and do no harm in the other areas.

Experience

Learn From It

It's easy to Monday morning quarterback a failed startup. I've often heard the query "What were they thinking?", but sometimes this is not really a fair question. The reasoning may have seemed sound at the time - certainly if someone provided funding, then the idea must not have been an obviously terrible one.

> The first startup I joined had a pretty reasonable-sounding plan: sell a full-featured 3D graphics content creation system to game developers, based largely on technical suitability and underpricing the twenty-thousand dollar per unit competing products. Who would've known that Autodesk would soon enter the market with a vastly cheaper product and established presence in the game market? I didn't.

Beware the entrepreneur who only knows (or remembers) success and thinks he has the Midas touch. success with a startup is largely the luck of the draw. Failure, on the other hand, is not a bad thing - you can learn from failure. But some entrepreneurs are repeat offenders.

> I worked for one CTO who exhibited many admirable traits, including loyalty - but to a fault. I was a beneficiary (and much appreciated it), but one of his golden boys returned the favor by alienating the rest of the staff and prompting several senior engineers to leave. (I was promoted quickly in that job) The attrition problem was eventually "solved" by putting the programmer in charge of another group, essentially transferring the problem and giving him a promotion to boot. After the startup was ignominiously absorbed into an acquiring company and quietly extinguished, the CTO chose to start another company with, guess who?

Show me someone who's running a startup, and I'll show you someone with an overabundance of self-confidence and/or someone who's tired of working for other people. Either way, recognition of past mistakes (unless they're made by others) is often not part of the package. But if you're going to go through the school of hard knocks, you might as well learn something.

Use It

Especially in a startup, you need to take advantage of your staff's existing expertise. Surprisingly, small companies can be as clueless as large ones at recognizing individual assets.

> Despite mentioning repeatedly that I'd worked on military visual simulation projects, the computer graphics company I joined never bothered to consult me when they tried to get into that market. For a big corporation, that's somewhat understandable (although such a corporation should have the resources to set up an employee database to check for in-house talent). For a startup in which budgeting and time-to-market is crucial, there's no excuse - just send email to your staff. How hard is that?
>
> I spent interminably painful weekly meetings in another startup in which the CEO would raptly listen to the CTO expound on topics he knew nothing about. I'm certain better answers about machine vision and image processing, for example, could have been obtained from the engineer who did his PhD in machine vision. Given that I'd just left a wireless Internet company, it would have made sense to ask me about wireless devices and operating web servers. (They almost tried to invent their own secure communication with web servers, not realizing there is a standard used for e-commerce).

Even if you stick to your core competence, you're going to run into unfamiliar customers and application domains. Before running blindly into them, query your staff - "Does anyone know anything about this/them?"

Focus

Be Cheap

There used to be a saying that Silicon Valley high-tech startups would decline once they moved into new glamorous digs. That saying referred to high-profile companies like Silicon Graphics, but I've seen this happen even in early-stage startups.

> When I joined a San Francisco wireless internet startup that had just received its Series A financing, they had just moved out of one of the founders' homes and into a moderate-sized office in the Financial District. In three months, they took a long-term lease on half of the top floor (where I had an amazing view of the bay), started purchasing expensive office furniture, and hired three administrative assistants, for a staff totaling no more than twenty. When the next series of financing looked less inevitable, the COO had to clamp down on spending and we ended up with twenty dollar utility tables as desks, to complement our eight-hundred dollar chairs.

> The next startup I joined was based in artsy Venice, CA. So of course the office upgrade involved a complete redesign by an architect involving curved walls and all attached desks and counters had to have matching custom curves. A Feng Shui consultant then walked around advising how to maximize the good fortune of the space. To no avail, as there were layoffs within a month after the work was finished.

Don't spend money that you don't already have.

But Not Too Cheap

On the other hand, don't pinch pennies at the expense of getting things done. Starting with people.

> I was offered a bit of sweat equity to take part in a startup but bailed out quickly when I learned everyone else also had a day job, thus turning it into a fun activity in which participants could play the startup game and brainstorm cool business ideas, instead of an urgent mission. If the founders had paid a full-time contractor they could have had a prototype running in a month instead of blowing at least that much over the course of a year with nothing to show at the end.

And there's no point in investing in people if you're not going to give them the tools to get the job done right, and on time.

> One company president who was ready to hire more engineers to accelerate my project but unwilling to buy more computers. While he wheedled our partners to loan us machines, I had to tussle with the documentation and training groups over the existing workstations.

The saying goes, time is money, but in a startup, money buys you time. Don't waste it.

Start with One Thing

I've never seen a startup that tried to begin with several products at once succeed. You can be GE later, but start with one project.

> One startup that I consulted for was so optimistic they rented manufacturing space and purchased booth space at CES before they had connected a single wire or written a single line of code. They could have implemented a prototype within a few months just using stock PC hardware, but instead got distracted by different ideas, ranging from robots and educational software to digital content distribution. In the end, they had nothing to show. Another startup that I helped get running also had eyes too big for our stomach. We started developing two rather ambitious computer products and didn't completely follow through on either.

And Finish It

The easy part of a startup is the starting part. Finishing something is the hardest part. Being the idea guy is fun (consulting is great work if you can get it), but developing, debugging and polishing a product, and then deploying, supporting and maintaining it, is painstaking work that requires tenacity and discipline.

> At the computer graphics startup I helped form, I don't know if we could have sold anything, but certainly we could have had a usable product if I'd bothered to implement file save/export capability. As it was, I left it a demo, and one of the few, and regrettable cases, where I can't point to a finished product. And when we abandoned development, testers who wanted to use our product, couldn't.

So whatever product or service you decide to start out with, get it done before you move onto something else. You need to prove to yourself and to everyone else that you can execute.

Look Ahead

Growing Pains

My favorite time to join a startup is when they still have less then ten people - at that time, everyone is still focussed on just getting something going and seeing how it turns out. Once the company headcount reaches fifteen, diverging agendas involving turf issues, managerial rivalries, and career ambitions rear their ugly heads.

> My first job at a startup was chaotic but fun until the headcount reached about twenty - then it seemed everyone had a management title and turf to protect (including me). That was also my first management position, and I was surprised to find that dealing with the engineers in my group was the easy part - dealing with all the other managers was the hard part.

Plan ahead of time how you're going to handle the personality conflicts, communication issues, and different requirements of a larger and more normal workforce.

Keep It Clean

It's all too common to see stories of malfeasance in large, publicly-traded corporations, and the natural reaction is - how could that happen in a professionally-run company? And why would wealthy executives take the risk of bending the rules here and there?

Well, that kind of stuff happens all the time in small companies - it just doesn't get the same headline publicity, unless you watch The People's Court. Big companies were once small companies, and big-time executives caught with their hands in the cookie jar used to be small-time players looking for an angle.

> Backdating? Clients routinely take their time putting together contracts and then expect me to backdate them. One former employer contacted me years after I left and asked me to sign and backdate an employment agreement to satisfy a potential acquirer. Stock pumping or fake articles using false identities on the Internet? I found a rave review of a mediocre game that I helped develop at a small game company - it was obvious from the crude writing style that it was written by the company president. And I've seen some other stuff that looked so fishy I don't even want to comment on it....

A lawyer for one of my startups asked us if we wanted to do things "clean" or "dirty" - that shouldn't even be an option! Don't get started with bad habits - if you're considering something that would get you in trouble in a larger company, just don't do it.

And it's not just a matter of staying out of jail. A dicey reputation is a lot easier to pick up than to get rid of. Whenever I have a prospective client or vendor, I ask around - sometimes the answer is "run the other way". And the way you conduct business will become ingrained in your company - that can come back to haunt you.

> One marketing guy reminded me of the pathological liar character on Saturday Night Live. He prevaricated like other people breath. He told me he was an RAF jet fighter pilot. He informed another coworker that he held a patent for a particular printer technology. He said he needed time off because his wife was pregnant (not). And then he skipped to a direct competitor.

No one wants to deal with a company that can't even trust its own employees.

Have an Exit Strategy

As in war and casinos, you should have an exit strategy. Are you aiming to cash out via an IPO or acquisition by another company? Do you plan to run this company indefinitely? What are you going to do if the current product or strategy doesn't work out or if you fail to bring in enough revenue or financing? Change strategy? Declare bankruptcy? You should have a Plan B whether it involves cashing out on top or cutting your losses.

What I Learned@MIT

When I get together with my old college pals, we typically regress and complain bitterly about the lack of women (this was back in the eighties), the concrete surroundings and lousy weather, how we missed out on the fun we're sure everyone else had at other colleges, and finally how they didn't teach us really practical things. Mostly we complain about the women. But as far as practical knowledge, looking back, I can see there were plenty of unintentionally imparted lessons about real life in the engineering world.

Work Alone

My software engineering course, 6.170 (MIT classes are known by number rather than name), consisted of successively more complex projects, starting with small modules with well-defined interfaces and culminating in three-person projects where we would partition the application into modules assigned among us and then put everything together for a demo (and our final grade).

Our project that year was a reversi game - my teammates worked on the user interface display and keyboard input, while I worked on the actual game engine. My teammates alternated between sending me pointless talk messages in the lab (for you youngsters out there, the forerunner of instant messaging), complaining about my absence when I wasn't in the lab, and then covered their own asses when our program displayed an obviously incorrect board in the demo ("See, the user interface is working, his game engine must be the problem").

One could draw an important methodology lesson:

> Integrate early and often.

But I think there are some overriding lessons in this case:

> Academics don't know anything about software engineering. Setting aside the faith in the design-partition-integrate process (isn't that cute!), using a research language only used at MIT on an obsolete operating system (TOPS-20) on slow machines in an overcrowded lab isn't a recipe for productive software engineering. Although that's probably how we ended up with the current state of the air traffic control system.
>
> Don't work with anyone else if you don't need to. I could have done the entire project myself in less time (and I did, for my bachelor's thesis - a Reversi program running on an experimental multiprocessor machine).

Keep it Simple

As the final project of my electronics lab course, 6.111, my partner (who went by the moniker Psi) and I wanted to build a computer. My partner wanted to build a Lisp machine - the Lisp machines of the time involved a lot of memory and software, typically selling for $100,000, so I persuaded him to scale our ambitions down to a Forth machine.

But then our teaching assistant looked at our design proposal and felt it wasn't complex enough and required us to double our chip count. After sleepless nights burning programmable logic arrays, finding/returning/replacing faulty chips, we had a wiry mess of lights that occasionally blinked out a correct computation.

The engineering lessons learned:

> Watch out for long wires - the digital abstraction doesn't always hold when inductance gets in the way.

The real lessons:

> Establish a good relationship with the suppliers. The old men staffing the lab parts department would give the best, i.e. working, parts to the pretty girls ("she has better parts", one of them replied when asked why he gave atypically good service and components to one of the female students), and recycle the burnt out chips among everyone else.
>
> Keep the design as simple as possible. Adding complexity just for the sake of it is just risk maximization - not something you want to do in real world engineering projects, unless you're graded on complexity and not a working result.

Keep It Working

I did build a functioning computer in the undergraduate computer architecture course (possibly my favorite), 6.115. The course progressed through the construction of a computer from scratch - and I mean really from scratch, building the processor and clock, wiring the data pathways, and writing microcode.

The end of the project featured optional entry in a performance contest - if your computer is the fastest, you get to keep it (worth a couple hundred bucks). The hardware guys could add optimizations like caching - I restricted myself to the many obvious optimizations available in the microcode. But even with software changes, there was a long turnaround interval in getting them burned on the PAL's. In the end, I was unable to enter because my changes were buggy and I didn't have time to debug-fix-burn-test.

My hardware engineer friends might say I should have left the software alone and worked with hardware, but the fact is, only a few people entered the contest with working kits, and only the winner had achieved a significant performance improvement. So the lesson learned is:

> Keep your system in a working state. If I had made one minor change per PAL burn instead of trying to make every optimization I could think of, I

> would have entered the contest with a handful of optimizations and at least placed second.

Steady Wins the Race

I was a crunch-time student (which, ironically, has prepared me for common industry worst practices). It seems in hazy retrospect that I stayed up every night cramming. The result was academic probation my second year followed by mediocre grade points the rest of the way.

I noticed the students who had perfect grade points weren't the whiz-bang smartest - they were the ones who kept regular schedules and maintained discipline, closing their dorm room doors to study, eating regular meals and going to sleep at midnight.

> So I wouldn't go so far as to say slow and steady wins the race, but steady is important.

Don't Work from Scratch

I tried to do all my homework ("problem sets", they're called at MIT) by myself. Mistake. There's a time-honored tradition of, shall we say, collaborative work among MIT undergrads, assisted by class "bibles" containing assignments and exams material, meticulously assembled and lovingly passed down over the years. When the professors, clued in or not, see their students handling the class material easily, they increase the workload (hence the on-campus popularity of the phrase, "drinking from a fire hose") until the curve is back down to where they like it.

> Trying to learn everything from scratch is a loser's game - take advantage of available literature.

This isn't to say you shouldn't learn the material - if you're going to apply the knowledge, you have to learn it. A fraternity member in one of my classes turned in a photocopy for his homework. Now, that's just too much.

Read the Documentation

My woeful track record on hardware projects over three years was compounded by my inability to use the ultra-cool and completely mystifying Tektronix oscilloscopes that populated the labs. I would twiddle the knobs helplessly until a TA would come by and click-click all of a sudden I had a nice waveform on the screen. When I wrapped up the final project of my final lab in my final year, I happened to look behind the instrument and realized each of them had a manual.

> RTFM

Remember Presentation

When I was there, the undergraduate MIT body boasted a three-to-one male-female ratio, a marked improvement over previous years (and vastly superior to Caltech's nine-to-one ratio, which scared me away from that school). I could count on my male friends to ditch me any minute they had a chance to talk to a female classmate. But on Friday nights they gazed turned to the buses from area all-female schools disgorging perfumed, coiffed high-heeled undergrads looking for MIT frat parties, while MIT women in jeans and t-shirts looked on in irritation.

When there's competition, packaging counts.

Keep Your Standards High

MIT does try to attract a diverse student body - at a party hosted by the chairman of MIT, a classmate of mine hailing from Michigan asked a representative from the admissions office if MIT gave preference to applicants from underrepresented regions. The rep explained that MIT gave consideration to the educational obstacles Midwesterners might encounter.

Notwithstanding that politically-correct response (or else the admissions officer really thought I studied by candlelight after putting the hogs back in the pen during my high school years in Iowa) you can't get into MIT, and you definitely can't get out, unless you can get through the math and science courses. And MIT doesn't have catch-up courses for varsity athletes or "physics for poets" that even the Ivy League schools have. You have to pass all the core courses that everyone else has to, and that includes the swim test. And you don't get in by being pretty - MIT doesn't require a photo with your application, unlike Stanford (and, oddly enough, CalTech - maybe it's a California thing).

Furthermore, as an MIT undergrad, you get very bright peers, famous researchers as your instructors (and some of them are excellent instructors, too), access to cutting-edge facilities, and opportunities to participate in research. Besides the brand-name degree, you come out of that institution with high standards.

I came to that realization soon after graduating and moving to Texas. The sales guy who sold me my first car at Toyota of Irving asked me if MIT was in Minnesota. Slightly more informed was my coworker at Texas Instruments who stated upon meeting me that he would have gone to MIT if he had known about it. (I thought that was the dumbest thing I'd ever heard, but it was my first day) And a few years later, I was astounded by the level of whining I encountered as a teaching assistant at Johns Hopkins for Computer Literacy 101 (the name says it all). At MIT I listened to braggadocio about consecutive all-nighters and a lot of persistent negotiation for grades, but the elite students of JHU were upset at me for not announcing the exact questions to expect on their exams.

Even when your expectations are low (and to be pragmatic and realistic,
they often have to be), maintain high standards.

Smart People Are a Dime a Dozen

At MIT, everyone is bright, and some are super-bright. Some super-bright people are total jerks, and some are the nicest you'll ever meet.

I had one famous computer scientist as an instructor who bore down on a student for not knowing how to solve a digital signal processing problem - after she protested that she didn't know the complex algebra required, he said, "come to me after class, and I'll teach you complex algebra." And then, after class, he told her, "I don't have time."

In contrast, I had an equally famed materials science instructor who set aside office hours to meet with any of his dozens of students who needed help on their homework, even though he had plenty of teaching assistants.

> So when I join a company and hear how smart the people there are - I'm not impressed. Unless they're also good people to work with. That's a lot harder to find.

Theory and Practice

Despite boasting some of the brightest minds and a top-ranked business school, MIT is hardly a flawlessly-run institution. How many other industries can get away with increasing prices faster than the rate of inflation? Any illusion I had that MIT was a smoothly-running machine was shattered when, as a part-time system administrator, I had to run a purchase order by four consecutive desks in the purchasing office, where the form was literally rubber-stamped each time, only to return across campus to my supervisor, who looked at it and said, "this isn't right".

> You can have smart people, lots of money and the latest technology, but your execution can still suck.

And like any other bureaucracy, MIT seems to run smoothly most of the time, but when something goes wrong, fixing it is like arguing with a rock. I didn't get paid the entire summer I worked at the MIT Microcomputer Center because my manager repeatedly submitted my monthly timecard late (and lied about it), and that gets you shrugs instead of paychecks from the payroll office (I got paid after complaining to the Information Services director that I had tuition to pay, and my boss got a nice referral to a cushy job at another high-profile university) When I later worked in one of the MIT labs, it was FedEx who had a similar problem - they refused to accept deliveries from us for a while because MIT's account was in arrears.

> Top-notch institutions attract the best and the brightest, but that doesn't mean they don't also attract the worst.

My favorite encounter with mindless MIT bureaucracy came after graduation. The student loan department sent me a letter saying they didn't have my address. (I suppose I should have called them and complained I didn't have their phone number). The humor of the situation was diminished by the late fees they charged me.

Don't Trust Politicians

Before graduation, some of my classmates in the student government went around asking for donations to our class gift, a scholarship fund. They neglected to mention that only members of the student government or varsity sports were eligible. Members of the unique and probably more useful organizations like the Lecture Series Committee (they arranged popular weekly showings of modern and classic movies and hosted such distinguished speakers as William Shatner), were left out.

> Even at MIT, nerds get screwed. Or, as they say in New England, scrod.

Agile Isn't

"The patient died, but the operation was a success."

What's Wrong with Agile

A director of engineering once characterized me as "in-the-closet" when it came to process. I do think there are important principles and procedures to follow in software development, but when I hear people brag about having a process, I shudder. Agile development is no exception.

I liked agile development...before it was called Agile. Now everyone says they're doing Agile, job postings list openings for "Agile/XP programmers", and Agile consultants and authors are everywhere.

It's a Religion

Because Agile has so many fervent believers, it's the worst thing to happen to software development since Microsoft. When optimizing programs, engineers are supposed to know that little improvements here and there don't matter if you don't fix the bottlenecks. But given clueless management and a project without direction, somehow Agile is supposed to save the day by saying, don't worry about it. That's not a process for creating quality products, it's process as a coping mechanism.

Normally, processes are pushed by management consultants on gullible management. But Agile is a religion for the masses - rank-and-file programmers think it's their salvation. They recite scripture (design patterns), participate in rituals (daily standup-meetings, unit tests, pair programming), Bible study (reading groups), and community worship (discussion lists and wikis). As a reward, they are assured by the high priests (consultants) they are Good Programmers.

Agile is bottom up. You don't have to know the Big Picture. In fact, you're better off not knowing. Just go month-to-month, week-to-week, write little snippets of code with unit tests, and keep asking "is this what you want?" Forget design. (Seriously, if you had a design, what was it?) Give up on long-range planning. It's the type of fatalistic approach that appeals to those who have been burned and those who don't know any better. You will be rewarded in the afterlife. (or with stock options)

It's Manufacturing

Agile also appeals to management by encouraging the misconception (or wishful thinking) that software development is a form of manufacturing.

> One large software project I worked on had an architectural identity problem - if you looked at any piece of the code, it was hard to discern the intent. Not because the code was poorly written. On the contrary, all of the thirty programmers on the project were highly qualified. The lack of design continuity arose from the practice of assigning the tasks pinned up

> on the corkboard to any available body. I was assigned several user interface bugs because the regular UI programmer was busy on other tasks. This had the pleasing effect of marking all of those bugs as "in-progress" or "fixed" in short order. But since the regular UI programmer had to inspect and approve my fixes before they went in, and he ended up rewriting them later, it really cost more time and effort in the long run.

There's a saying in video game design, "Find the fun". Squeezing software development into a manufacturing mold could be termed "Find the fun, then remove it". Software development is not manufacturing, unless your idea of manufacturing is designing a new engine for every car that comes off an assembly line. Software is custom engineering and still resembles a craft more than anything else.

It's Process for Process Sake

Naturally, manufacturing companies would most like treat software development as manufacturing. They love processes and "best practices" - TQM, ISO 9000, Six Sigma. They also have the worst software development.

> I worked for a factory automation company that regularly flew our Director of Engineering out to meet their Software Best Practices committee. Yet we had a codebase that was not buildable for days at a time, misguided design, dysfunctional and hostile interaction among engineers, product requirements appearing immediately after product delivery, and components that were untested to the point of being obviously uninstallable (if anyone bothered to just look at it). And that was our cutting-edge Java application - typical start of the art in manufacturing was Visual Basic.

I often treat "process" as a dirty word, but process should be a good thing. Learning is a process. Self-improvement is a process. Doing things better every time around is a process. Process isn't something you trumpet - good people (in both the moral and competence senses) don't talk about how good they are, they just are. In fact, people who like to talk about how good they are usually aren't.

> One of the more entertaining workplaces I've seen included a programmer who evangelized Agile and a manager who pushed ISO 9000. The programmer covered his cubicle wall with scrum notecards and kept complaining we didn't have proper sprints yet didn't maintain a reproducible build process and ignored all feedback on the product from management. The manager wrote a process manual that didn't say much except that they had a process and gave his employees pop quizzes on the manual. Yet he didn't bat an eye when distributing release CD's filled with random builds of the product placed in random directories. Those guys

> made a perfect couple. In the meantime, there was plenty of work for service engineers.

Indeed, one of the tenets of Agile states "people before process". And defenders of Agile will immediately respond to any of these criticisms of negative anecdotes that "You're not doing it right." (This would be more credible if claims of Agile success were examined as skeptically) But theory doesn't matter if you have to suffer the practice. When people talk about process, what they really mean is a formula that allows them to avoid thinking about principles.

Striving for Mediocrity

The public schools system is great at educating the middle. If you're slower than the middle, you get pushed a bit or repeat a grade. If you're ahead of the curve, you just get bored.

Popular agile practices are also geared toward the average (or is it the median?) Scrum meetings by definition require everyone to move together (ostensibly like a rugby team moving up the field together - or like a soccer team of five-year-olds moving en-masse around the field while their parents yell encouragements). Pair programming is great for those who need to have their hand held or code monitored. Proving that one plus one is sometimes less than two.

> I worked for one company that boasted of its agile practices - in particular, scrum. Overall, it was a quality project with high production values. But I found the process more a hindrance than a help. The daily meetings just meant there was no planning - instead of working out task dependencies early, developers just waited until the morning scrum to say, "oh, by the way, I'm waiting on this..." The pair programming required for code checkin either meant someone was totally bored while I explained the code or I ended up quickly changing the code to match someone else's idea of good programming style just so I could check it in.
>
> The real reason it was a quality project was the fact that they recruited talented people who were able to work with each other outside of the process enough to get the job done during the crunch that the whole process was supposed to avoid. All the process accomplished was to give the project managers the illusion that they were in control of things.

Coding is not inherently a team sport. Doubles tennis can be fun to play and entertaining to watch, but the game is meant for individuals. The Cohen brothers can write scripts together and finish each other's sentences while directing, but that is a rarity - most good work of that nature is performed by individuals. Your best programmers are the ones that can get the project done. But corporations feel more secure with all their engineers contributing mediocre work than with a handful of engineers contributing superior work and the rest killing time.

LEARN FROM OPEN SOURCE

Agile arguments tend to be anecdotal (like those of yours truly): "We used it on Project X and it worked." But if you want case studies, why not look at open source projects? One could argue that open source development is the most agile development of all.

Transparency

More than any other type of project, open source projects are transparent. You can evaluate for yourself the quality of the code (just look at it), the popularity of the product (e.g. SourceForge ranking, or in the cases of projects like Apache or Firefox, it's well known), and the level of development (check the history of change, who changed what and when).

Communication

With transparency and the tools to support it, communication follows. A typical open source project has active discussion lists (email and/or forum), a wiki for collaborative documentation, occasional chat sessions for real-time meetings, and occasional face-to-face get-togethers at conferences. Code changes are automatically announced on commit mail lists, giving everyone an opportunity to inspect the diffs.

Testing

Also with transparency comes testing. Anyone who's willing to download and install is a tester. Again, tools are important. Users are more likely to submit quality bug reports (as opposed to just sending whining email) if the project web site sports a convenient bug entry page and searchable database. And users are more likely down to download the software, even free software, if the project has automated builds and tests.

Meritocracy

Contrary perhaps to public perception, open source projects are not uncontrolled pools of code. (It's not Wikipedia!) Successful open source projects are led by programmers with a firm direction. The set of developers allowed to commit changes is typically limited to a few trusted people. Some projects have elaborate procedures for admitting new committers. Developers have to prove themselves before getting the keys to the vault. Bad code submissions are ignored - not incorporated into the codebase just so they can be fixed later.

No Interference

With good tools for communication and good people doing the work, there's no need for constant supervision by professional managers. No need for frequent meetings. I have yet to see a notable open-source project with strict style guidelines. The development drives the process, not the other way around.

Competition

What motivates open source developers to maintain effective practices? Competition. Even more so than with commercial software. Release buggy software, wait too long between releases, take the product in the wrong direction, anyone is free to take the code and spin off a new project with a new group of people (or even some of the same ones). Open source projects compete in a free and open market for users and developers. The agile movement, on the other hand, started in large corporate projects that can live or die based on reasons that have nothing to do with quality.

Keep It Simple

Good tools. Good people. Transparency. It's not that complicated.

QA for Everyone

> "Is this going to be a standup fight, sir, or another bug hunt?" - Marine Private Hudson to Lieutenant Gorman, in *Aliens*

Test Early

Testing isn't something you do when you're almost finished. It's something you do as soon as you have something running. It's not just management that is prone toward delaying testing - programmers are often just as bad.

> On my first commercial software project, the software was so incomplete and crash-prone that I was reluctant to give the first "alpha" build to our test group. Fortunately, the QA manager insisted, and the resulting bombardment of bug reports forced me to immediately focus on the problems that prevented the testers from exercising the application in any meaningful way. Left to my own devices, I would have worked on what seemed to be the harder and more important core 3D issues, and probably delayed too long on seemingly mundane issues like the user interface, load-save functionality, and compatibility with all the varieties of consumer hardware we were planning to support.

The bug count on any project has to go up before it goes down. Get that spike early and you can whittle it down gradually, instead of attempting to deal with it at the end.

Test Often

Once you start testing, keep testing.

> On my first console game project, our publisher didn't have QA department look at our game until just before E3, the big game industry trade show where retailers get a look at the upcoming games. That bit of testing just provided foreshadowing of bug reports to come - after the show they didn't look at the game again until less than three months before the submission date. Within a week we had several hundred bugs logged.

Testing should be a continuous part of the development process, not a footnote.

Test Everywhere

Test on all the platforms that you plan to support. You can't assume that a product running on a system will run on all minor variations of that system.

> At a computer graphics software company developing applications for SGI workstations, our chief sales guy proudly informed me he sold thirty seats of our product to a prominent video game developer. Unfortunately, those were thirty licenses of a product that was still in beta, and the customer was planning to run on new SGI workstation models that we hadn't even seen, yet.
>
> I had to make two followup visits to the customer - one to make a feeble attempt at diagnosing our product crashes on the new machines, and the other to get yelled at by the customer (they were yelling at one of our sales support guys, but he couldn't take it any more so they sent me in as a replacement)

In an ideal world, everyone would have access to the every platform of interest. Usually, there will be compromises due to constrained budgets or limited availability of the target hardware, particularly if you're in a small company. I've seen several situations where the QA group was first in line for all new types of hardware, probably reasoning that QA is responsible for testing on all platforms (turf may have something to do with it, too). But assuming there's logic behind it, that logic is misguided.

> Soon after I joined the aforementioned graphics software company, the QA manager chastised me for not testing my code before I integrated it into the test build. Well, I did test it on my workstation, but it happened to fail only on hers because it was the latest and greatest model and she had the only one (This was at a time when it seemed like SGI was cranking out new models every couple of months, each with new subtle hardware and software incompatibilities) This meant I had to sit at her desk to try to diagnose the problem.

Every programmer knows it's ridiculously more difficult to track down and fix code on a machine that's not set up with a development environment. So you can save everyone a lot time and grief by making sure the programmers get first dibs on new hardware (as long as they use it!)

Test Everything

There's not much point in just testing the obvious - test what the developer may have overlooked. Exercise boundary conditions (zero dollars, empty text values, empty files), feed the program garbage data (negative numbers, text instead of numbers, non-alphanumeric characters, random files). Treat it like a game, where the goal is to find a way to break the program.

Test Methodically

It's just as important to know what works as what doesn't work. So record all test results. Better yet, write down a test procedure. Even better yet, automate testing as much as possible.

Not only does automated testing, in the same manner as automated build and backup procedures, give you confidence that you have a known and reproducible process, but it allows you to test conditions that are impossible to create manually.

> Automated testing is especially important in simulating network usage. When I worked on a networked virtual military exercise, manual testing revealed a lot of interesting bugs (submarines flying above water, ships plowing through continents, planes flying backward), but problems with handling high network traffic didn't become apparent until the exercise day when I turned the switch on and our whole system crashed.

> Web-based systems in particular should always have automated test systems. A lot of web startups see their products work with a few users and just extrapolate. I was horrified, after moving from the Bay Area to a startup in Venice, CA, to see the web server team test by asking a few employees to try connecting to the server at the same time. And I was more horrified to see that test crash the server immediately.

Bug Reports Welcome

Testing is everyone's responsibility. Hopefully everyone involved in developing a software product will actually run that product. So take advantage of that manpower by making it easy for anyone to submit a bug report.

> When I worked with Silicon Graphics as a vendor, back when they were top dog in the graphics world, I had a love-hate relationship with them. Mostly the latter. Their developer support program was great, unless you wanted to report a bug. They officially could not receive bug reports - those had to be submitted to their (paid) customer support program which only accepted problem reports over the phone, via a receptionist who would take down your phone number and workstation system ID, and attempt to jot down the description of your problem. Weeks later, a junior engineer would call you back so you could repeat the problem description. Then the engineer would be transferred to another group and another engineer would call you.

It's one thing to solicit bug reports from everyone, it's another thing to actually get them. Only the truly motivated will report bugs if the process is at all inconvenient.

> I worked with one QA group that put together a sorely-needed online bug entry form. The bad news - it looked like a tax form. Field after field, all mandatory and without convenient default values. The result, hardly anyone outside QA would submit a bug report, and any that were submitted tended to have misleading wrong data.

Keep the bug reporting process simple. It's a bug report, not a TPS report.

Show Me Yours, And I'll Show You Mine

By the same token, use your customers as testers. Companies often do this with beta and even alpha releases, but there's no reason to stop there. Assuming you have a real customer base, the real testing will start when your product ships.

> I wasted several months fretting over a mystery bug when developing a product for SGI workstations and ended up releasing the product with a convoluted workaround. After several months of complaining alternately to our compiler vendor and to SGI, and then tracking the behavior down to a particular ordering of the libraries passed to the linker at build time, our SGI support contact informed me that it was a known bug that had been on the books for a year.
>
> In contrast, one of my favorite experiences in adopting Java has been the Sun Bug Parade, which has allowed me to check if a bug I've encountered has been encountered by others, and what the current status is. As a bonus, the Bug Parade allowed you to vote for the bugs that you consider most critical and displays the current vote for each entered bug.

Protect the Database

QA has their own product - the bug database. The bug database represents the state of the product and should drive all development decisions. Thus it should speak the truth.

> A month before the release date of a 3D graphics program, the company president insisted that VRML support had to go into this release ("I just want it", was his cogent argument). After we wasted a couple of days trying to cram it in, I pointed out that our bug database listed 300 bugs, most of them categorized as high priority. The president said "Oh, I didn't know we had so many bugs" and changed his mind.

Just about every QA group I've dealt with would escalate originally low priority bugs to higher priority toward the end of a project as a not-so-subtle way to tell developers what to fix. That is a misuse of the bug database - if all the critical and high-priority issues are resolved, don't relabel low-priority issues as critical. Just say it's OK to fix low-priority bugs, now. If management has prohibited work on low-priority bugs, then fix that (at least the edict, better yet, fix management). And don't downgrade a bug just because no one wants to deal with it.

> The grandest act of wishful thinking I've seen applied to a database was invoked by a game publisher who ordered their QA staff to mark all remaining bugs in the database as fixed after we made what we hoped was our final submission. As if the bugs would really just disappear just because the product was going to ship. The tactic made even less sense considering the European version of the game still had to undergo localization, testing and submission.

Cooking the QA books is not as illegal as cooking financial books, but it should be. Protect the integrity of the database.

You're Not the Boss of Me

I've met a few QA people who somehow thought that QA was on top of the organizational pyramid.

> When CMM was all the rage, a QA coworker of mine gave a presentation on the topic and concluded that his group should oversee all of development, meanwhile waving the CMM book that clearly stated that shouldn't be the case (there should be CMM oversight of both groups, but considering we had fifteen people, I don't know why were talking about it at all)

Most QA professionals I've met, depending on the industry, were in it as an entry level job (in game development, it's touted as a first step to becoming a game designer) or as a less stressful alternative to programming (one tester told me she just didn't like thinking about code after coming home from work - I don't blame her!). But it's the type of job that can attract cops and bureaucrats.

- I worked with one QA manager who, while a nice guy, liked to lecture the development staff about proper procedure. It surprised no one when he casually mentioned that he used to be a traffic cop.

Now, I love working with testers who want the product to be the best possible. But the best way for QA to estrange the development staff is to take things personally.

- On the first project where I was a manager, the QA lead was originally going to assign a senior artist to test the graphics product we were developing, on the theory that experienced professionals in the field were best acquainted with the valuable features of the product. That might have worked out OK, but I'd seen enough problems dealing with some senior (and even junior) testers who were accustomed to thinking of bugs as impediments to their own pet art projects (one guy's bug reports consisted of "It's broken", "I didn't do anything", and the ever helpful "Fix it!") rather than behavior they should attempt to discover, reproduce and track down. Since I was more interested in getting quality bug reports (or at least peace and quiet) instead of whining or tantrums, I almost begged the QA manager for the services of one even-tempered entry-level hire. I think that paid off - he did a great job sitting through crash after crash and submitting dozens of detailed bug reports every day until the code got stable.

Remember, as a tester you're paid to find and help track down bugs. Disgruntled, whiny customers can be had for free.

But You're Not Peons

On the other hand, management needs to understand that QA is an intrinsic part of software development, not a cleanup crew shuffled in at the end of various projects. Especially, when you have a salaried QA group.

- While working on a long-term hugely spec'd proposal processing server for the Hubble Space Telescope, I was both bored and stymied (I'm not proud of this) and instead spent my time on a little stealth project, a GNU Emacs mode for editing proposals. It attracted some interest from in-house users and management, so I trudged down to the QA room, where I found everyone was sitting idle (as you may guess, we weren't cranking out a lot of software). The lead tester showed some interest in trying out my program but said he couldn't spend any time on it without permission from his boss, then returned to his solitaire game. Your tax dollars at work.

It's not just a matter of keeping people busy. QA knows how to use your product as well as anyone else in the company, and probably better. Use that expertise.

- Sometimes it seems people want demos to fail. I've had several managers who habitually dropped into my office expecting impromptu demos for a VIP right while I'm coding. One was apparently unaware that I arrived at work three hours early to prepare a demo and decided to bring the visitor back a few hours later so he could watch me compile and crash. I got so irritated at another that I let him run the demo himself and flail in the debugger. In each case, the QA guys were in the room next door. Not only were they painfully aware of what potholes to avoid if they wanted to try out various features ("don't click there!"), they also knew which build from the past several days would demo best.

But testers aren't just useful for demos - they are your experts on the state of the product on the state of your product. If you want to know how close your product is to release, ask them.

- The one project where I had total greenlight authority, I consulted not upper management (who kept demanding new features), or marketing (who wanted to delay the release date for no good reason), or sales (who proudly sold a site-license for an unfinished and crash-prone version of our last project to a high-profile customer) - I asked the guy who'd been testing it every day for six months. If he said it was ready, it was ready.

Work Hard, Play Hard

As a break from the usual advice on how to make your project run like a factory or how to impose your will upon the troops, here's an idea - run your project with game design principles in mind. This article is targeted toward game developers, but it may be generally applicable.

Game Design Your Project

Game development is the software engineering equivalent of an extreme sport - the combination of fast-moving technology targets, the complexity of integrating design, art, audio and software into a single production pipeline, the demands of publishers (sometimes reasonable, sometimes not) and the typical "death march" schedules should dissuade anyone who wants a normal life from joining a game project. And it doesn't pay startlingly well, at least compared to other forms of software and media production.

So why would anyone want to be a game developer? And how do you motivate and retain game developers once you've got them on your project?

Well, game companies typically list "a passion for games" as a prerequisite in their employment ads. Perhaps this is the key - if game developers are also gamers, we may be able to apply game design principles to attracting and motivating game developers. After all, with game projects, as with games, we want to attract participants, inspire them to work through and complete the endeavor, then come back for another round.

In this article, we see how far we can stretch this analogy by applying some game design principles to game project management. The principles are selected from the introductory chapter of Richard Rouse's book **Game Design: Theory and Practice.** They are listed in their original order and with their original titles and categories, but are paraphrased for brevity.

Why Do Players Play?

Our initial query was: What draws game developers to this industry?

This is an appropriate first question to ask when designing a game. What brings game players to video games over other forms of entertainment? In other words, let's figure out what are the unique offerings of video games so we can capitalize on them.

Likewise, game developers could easily be working in other fields, other types of software development or media production. So once we realize the unique attractions of the game business, we can leverage them.

Players Want a Challenge

Game players enjoy facing challenges in games, especially single-player games. These challenges entertain, provide intellectual stimulation, and teach lessons that can be applied to other problem-solving situations.

Developers also seek challenges. Not challenges in the form of how-may-bugs-can-I-fix or how-many-hours-can-I-go-without-sleep, but, as in games, challenges that are intrinsically enjoyable and result in knowledge that can be applied in the future. Programmers want to implement cutting-edge algorithms that exploit the latest hardware yet operates within those hardware constraints. Artists want to create the best-looking models and animation within their given asset budgets. Game designers want to create entertaining and addictive gameplay experience through story, AI and level design.

Developers also have a pragmatic motive for intellectually and creatively extending themselves. Career longevity depends on staying current with the latest tools and technologies (ask all the developers who were slow to make the transition from 2D to 3D). Game companies are constantly looking for experience with the newest generation of consoles, and developers are always eager to work on the latest hardware.

Players Want to Socialize

Long before their appearance on computer hardware, games have mostly been social activities. Social computer games range from Quake-style fragfests with players typing insults to each other in the heat of battle, to massively multiplayer persistent worlds featuring chat-style interaction.

In my experience, even single-player games take place in a social environment, either in taking turns, watching others play, and kibitzing. Certainly in the heyday of arcade games, you could see crowds gather around superb players. And a visit to any of the countless web-based game forums will show that gamers argue vehemently about the merits of their favorite games as much as any sports fanatic.

Likewise, many employees want a socially engaging workplace, where they can learn from their coworkers, banter and sometimes compete with them. Some companies have company picnics and Friday afternoon beer socials. Game developers have after-hours fragfests (which blurs the distinction between game development and game play even further).

Players Want a Dynamic Solitaire Experience

On the other hand, sometimes players like to play games by themselves, either because no one else is around, or the player wants a dynamic interactive experience without the hassles of dealing with (real) people.

Even in a social workplace, most employees need some time to themselves to get their work done. This is particularly true of programmers, who need to do a lot of thinking, (paired-programming aside) but everyone needs to don the headphones during some stretch of the day to get work done.

In a larger sense, the workplace also offers the developers refuge from the people of their "regular" lives - spouses, children, friends, pets. (I had one manager who, at 5pm, would start moving from office to office, ostensibly to talk with people, while his wife called random numbers to track him down.)

Then there are those who don't want their work hours to encroach on their non-work life. These employees prefer minimal distraction and regular work hours so they can get home in time for dinner with the family, attend evening classes, play in a rock band, or engage in whatever other activities that constitute their personal lives.

Players Want Bragging Rights

Playing to win means getting respect from fellow players and bragging rights over them. This braggadocio is evident in multiplayer fragfests, organized game competitions, and the high-score tables on game screens dating back to the early arcade games. This earned respect also includes self-respect - completing games and winning, even against the computer, can inject a shot of self-esteem into someone who might be athletically, socially or academically less accomplished.

Developers would like to brag that they worked on the latest smash-hit AAA title, worked for game companies that are household names, worked with game industry luminaries such as John Carmack or Will Wright. Even just mentioning you're in the game business is a good conversation starter just about anywhere. I've worked in half a dozen different industries, but only since I started developing video games (even lesser-known titles), have I gotten a reliably enthusiastic response from laymen, including the guy at Starbucks. When I was working on manufacturing software, on the other hand, talking about my work inevitably resulted in a glassy-eyed stare, even from fellow programmers.

Part of the thrill in working on a game is seeing your name in the credits. I don't really like the idea of displaying credits in a game - it certainly isn't a common practice in other types of software, and it can easily be seen as an unfair and arbitrary process (in the film industry, it has political to the point where there are union rules about film credits). But it does seem to provide motivation and is important for career advancement in the game industry (many game job ads list credits on shipping titles as a requirement).

Players Want an Emotional Experience

Compared to other media such as film and literature, games usually exhibit limited emotional range, but at the very least all games attempt to produce excitement and a sense of accomplishment in game players.

Game developers want to experience those emotions in game development, too. Ideally, developers should feel an increasing level of adrenalin rush and anticipation as the project nears completion, and the gold submission should be accompanied by a profound sense of achievement. Pace is important - the end of the project should be the most gratifying, not anticlimactic.

Good emotions: interest, confidence, anticipation, excitement, satisfaction. Bad emotions: frustration, nervousness, disappointment, anger, ennui.

Players Want to Fantasize

Games take players away from their normal lives by immersing them in fictional environments and circumstances. As in the movies, the game world is idealized - with the exception of The Sims, you're not bothered by household chores, taxes, and addressing bodily functions.

In the workplace, of course, you still have to deal with bodily functions and taxes, but a job still offers a world separate from the home life and roles and missions that are different and possibly more exciting than available in the personal life. To different degrees, companies are like role-playing games in assigning employees to different official categories - game designer, producer, programmer, artist. Startup companies with more loosely defined responsibilities allow employees to assume a "be all you can be" attitude.

What Do Players Expect?

Now that we've established what motivates people to become game developers, it's time to address the second part of our query. Once we've got a developer on our project, how do we keep him present, happy and motivated?

In game design, we have the same issue - once a player has decided to play a game, he has expectations that must be fulfilled for him to enjoy and complete the game, so it is important to identify those expectations, conscious or not.

Players Expect a Consistent World

Players expect their actions in a game to have predictable results. Not right in the beginning - the player has to experiment a bit, in the way that infants do learn about their environment, to understand how things work in the game world. Seemingly arbitrary cause-and-effect will discourage the player and give the impression that the game is rigged.

In a game development setting, motivational pats on the back and even concrete rewards like raises and bonuses will not have the intended effect if the developer cannot understand how to get those rewards. Even worse are unpredictable punishments - if working harder or taking extra responsibility results in censure, then the developer will adopt a fatalistic and passive attitude. In some extreme cases, I've seen employees refuse to do anything until explicitly given an order or perform deliberately bad work.

Players Expect to Understand the World's Bounds

In games, players expect to recognize boundaries on actions and movement. Visual cues such as walls and precipices indicate the world's physical boundaries. The available controller actions constrain physical actions, e.g. some games have no jump button.

In the workplace, the employee wants to know the boundaries, too (they may think they don't want to have boundaries, but at the very least, they need to know the ones that do exist). For legal reasons, corporations often communicate boundaries on acceptable workplace behavior through orientations for new employees, employment handbooks explaining company policy, and, when they're particularly nervous about it, various forms of "sensitivity" training.

But all the legalese is ineffective without visible enforcement. Termination or other corrective actions will signal to everyone that the offending behavior, whether it be sexual harassment or just talking back to the boss, is out of bounds. Letting the behavior go will either lead to ambiguity or signal tacit approval.

Boundaries also include organizational boundaries. Game projects typically provide the publisher with a project document listing key roles and responsibilities of the development staff. This type of documentation is also useful to the team, so they have an idea who's in charge of what and who reports to whom. Management often finds it convenient to keep these roles vague, which has the advantage of providing some flexibility, but doing this for political reasons will just result in confusion and recriminations.

Players Expect Reasonable Solutions to Work

After gaining some experience with a game, the player has idea how to solve problems in the game. The player will be frustrated and irritated if any reasonable solutions based on the gameplay so far turn out to be ineffective. So designers should take care to accommodate such solutions even if they are not the primary solutions intended by the designer.

Game developers also expect reasonable solutions to work (how many times have you heard, why doesn't this work?) Game development is often at the cutting edge in terms of technology and scale, so oftentimes techniques that "should" work, don't. Even, tools and middleware and equipment for game development are notoriously flakey.

So it is important to ensure that everything within control works like it should. The devkits provided by the console makers may arrive late and crash frequently, but the commodity hardware and software (desktop and server computers, email, backup software, for example) should be rock solid. Compilers, debugging tools, and game engine middleware are often inadequately documented and in a beta state, so the production pipeline and internal tools should be well documented. The smoothest-running game projects I've seen still had plenty of mystery code, halting production pipelines, and IT glitches like servers crashing, data irretrievably lost, and ill-timed upgrades during crunch times.

Players Expect Direction

A game should give some indication of the the player's objectives. Otherwise the player may roam the game aimlessly wondering what to do, randomly attacking objects, NPC's and other players just to see if something will happen.

Game developers can also roam around a game project aimlessly. In the worst case, they will break other developer's code and art, and bitch, moan and complain.

Everyone wants to know what the game is about. What's the story? What type of gameplay are we trying to achieve? How does it fit in and compete with other titles on the market? This is what a high-concept documents is supposed to communicate, and this document should be readily available in-house as well as distributed to publishers. If you can't convince your own team of the viability of the game vision, then it's much less likely you can persuade publishers, their marketing staff, and retailers to buy into that vision.

Direction is also provided by the schedule. The final release date and interim milestones, including specification of the critical features required at each point, should be clear, reliable and changed only in drastic situations that warrant changing the entire schedule. In other words, the milestones and release dates should not be moving targets.

Scheduling is often performed down to a fine-grained level, in some cases to tasking day by day by day or even by hours. But while some developers may require micromanagement, it is important to make a distinction between this kind of supervision and a global schedule that the entire organization needs to be working toward. A day-to-day or even weekly schedule is volatile - vacations, sick days, emergency bugs and demos, server crashes and other natural disasters happen. Some tasks take longer than expected and some turn out to be easier than anticipated, and sometimes it makes sense to reorder them, but overall they should average out to meet the scheduled milestones.

Players Expect to Accomplish a Task Incrementally

Players usually know the overall objective in the game but expect to achieve this objective via a succession of subgoals. This provides awareness of incremental progress and reassurance that the player is on track. Without this feedback, a player could go off course and not realize it.

Not only do game developers find completing subgoals more tractable than large monolithic tasks, subdividing large tasks is vital to risk management and project scheduling. Then progress can be measured and validated by monitoring the completion of these subtasks. When developers jump into general assignments such as implementing renderers, physics engines, AI without defining components and tests that can be completed in sequence, then such a project can drag on for months without visible progress, until it becomes apparent that it's going in the wrong direction or that no progress is actually being made.

Players Expect to Be Immersed

Obtrusive user interfaces and game glitches, particularly crashes, distract from the player experience. And a character that is difficult to control or unappealing will also prevent the player from comfortably playing that role and feeling part of the game world.

In a game development environment, you also want each team member to feel immersed in the project and concentrate on getting the job done without distraction. Bureaucratic and corporate artifacts should not intrude on what should be a project that is rewarding unto itself. For example, timecards and sign-in sheets, thick employee handbooks, administrative paperwork, will remind employees they are on the clock and working for "the man". Instead, the necessary evils of running a business should be kept simple and to a minimum, and the environment should exude the exciting aspects of the game industry - the office should have plenty of games, industry magazines, posters, etc.

As far as providing a suitably appealing and easily assumable character to play, this does have a counterpart in the game project, too. Each developer plays a role (sometimes more than one) - an enjoyable role will be played with gusto, a distasteful role will be dreaded and performed without enthusiasm.

Players Expect to Fail

Players want challenge, so naturally they expect to fail at some points in the game. Moreover, those failures should stem from inadequate or incorrect play, rather than "tricks" or "cheap shots" utilized by the game. And the game should start out easy and ramp up later in difficulty to avoid discouraging players before the reward of the gameplay becomes apparent.

Game developers are also in the business for a challenge (or at least they should be), so they cannot be held back by fear of failure. On the contrary, developers should learn from failure. Attempts to implement new algorithms, use new tools will almost certainly result in some failures, all as part of the learning process and should be anticipated in the schedule. (This area where breaking schedules down too far will diverge from reality - you don't know how many different implementations of say, a dynamic shadow algorithm, you might try, but you should know when it has to be completely done).

These setbacks are acceptable as long as they are natural byproducts of the learning process, but aggravating if they are imposed by outside factors. Unrealistic schedules and frequent crunch times will leave room for less error while simultaneously increasing the number of mistakes. Unreliable hardware and tools...

As with a game, a game project should start out easy, so everyone gets in the flow and understands the rules - how to work with the production pipeline, how to work with the rest of the team.

Players Expect a Fair Chance

Although players expect to fail, they also expect a fair chance. Ideally, a player should be able to make it all the way through the game on the first attempt if no mistakes are made. This means that progress shouldn't require trial and error - it should be possible to deduce a successful path through the game. If the player finds that the only way to progress in the game is through guessing from sets of random choices, then it will seem like a waste of time.

Game developers also will become frustrated if it seems they have no way to make decisions short of guessing.

Players Expect to Not Need to Repeat Themselves

Players get annoyed if they have to repeat any tedious or painful portions of the game. Hence the availability of game saves, and, in particular, checkpoint saves.

The most obvious analogy in game development is avoiding loss of work. Code, game assets, and even production documents should be frequently and regularly checked into a source control system. And everything should be backed up periodically, with the archive media stored off site and test restores performed to verify the backup integrity. This will seem obvious to some and extreme to others, but data loss due to accidental erasure, hardware failures, and absent or faulty backups is all too common.

Another interpretation of this game design principle is that team members shouldn't be duplicating work. For example, timely communication and visibility of the code base should allow programmers to avoid redundant work and encourage code sharing. The game design and requirements, production pipeline, and any project and corporate procedures should be documented and easily accessible to avoid inefficiency in explaining and learning.

Players Expect to Not Get Hopelessly Stuck

A game should not allow a player to get stuck in a position from which there is no chance to complete or win the game. For example, a player should not be able to jump into a while from which there is no escape, aside from quitting the game. Either provide a way out or put the player out of his misery.

Developers also resent ending up in situations where they can't succeed, and rather than hit the Quit button, they may just sit there, resentful and apathetic. If they feel they're faced with unfair expectations given unrealistic schedules, unbounded features, late or scarce tools and assets, then they won't even try. If a developer is just not capable enough to succeed, then there's no point in breeding resentment by letting him linger on.

Players Expect To Do, Not to Watch

Rouse opines that players want to play, not watch cut scenes. While cut scenes can be instrumental in communicating narrative and setting up new levels, the duration of cut scenes should be kept to a minimum. Games that rely on cut scenes rather than gameplay inevitably fail to keep the gamers' attention.

Developers don't want to sit around and watch a game being put together - they want to be part of the action. Everyone has opinions on games and would like to develop a game that they actually want to play. You can't have game design by committee, but soliciting ideas from everyone will make them feel like part of the creative process. Internal newsgroups or message boards can be used for exchanging ideas. Contents can be held for names. Songs voted on. Blog-style developer journals can be kept for historical and promotional purposes.

One way for developers to feel part of the game is to literally make them part of the game, by modeling characters to resemble staffers, incorporating inside jokes, recording project members' voices for voiceovers - all of these will personalize the game for them.

Players Do Not Know What They Want, But They Know It When They See It

Once a game has reached a playable state, it is important to test it with real players and gauge their reactions. Focus groups cannot be relied upon to make game design decisions - that would be too easy. Rather, it is incumbent upon the game designer to observe their reactions and use observation and experience to discern what is and is not working in the game.

Similarly, employees may not be able to articulate precisely what they want in a workplace or project. Putting up suggestion boxes and soliciting feedback in employee reviews may elicit some useful ideas, but you'll get a lot of advice on running the project that still doesn't necessarily fix problems that are disturbing them. There is no substitute for the scientific method - observe your team dynamics, hypothesize about what's fundamentally not working (or working), make corresponding adjustments and verify your fixes work.

A Never-Ending List

Besides playtesting and following the rules described above, a game designer can come up with any number of game design principles based on experience and personal tastes.

In project management, also, the best rule of thumb is to use one's own preferences. What kind of project do you want to work on? Why are you in the game business, what attracts you to a project and what would be your expectations? As they say, your mileage may vary, but if you at least design a project that you would want to be part of, then you have something that appeals to at least one known type of person.

Back it Up: Safe IT Practices

IT is getting harder every year, due to the march of technology, increasing spam, and more nefarious security attacks. But there are some basic principles in taking care of your organization's IT needs, whether it's a one-person shop or a huge corporation.

Trust, then Verify

Backups are like insurance policies - you don't realize how much you need them until you really need them. It's not enough to have a backup system - you need to have assurance that it will save you when you need it to.

> An owner of a small game development company assured me that all important data was backed up periodically. It turned out this didn't include our source code. Even after this was rectified, the backup tapes were never verified, and a server crash lost a month's worth of work just before E3 (the big trade show).
> Complacency isn't limited to small companies. When I worked at a large research institution that operated the Hubble Space Telescope, the army of college students running nightly backups looked reassuring until one of my coworkers lost a file and found the backup was unreadable.

It's not enough to just make the backups. If you don't periodically verify that you can restore from the backups, there's not much point. In a similar vein, keep your backups offsite. If your office is ransacked or goes up in flames, the backups won't help if they happen to be sitting in the same room.

> This was my first practical on-the-job lesson. An experienced, high-level engineer was brought on to my project to help get things organized, and the first thing he did was shop for a fireproof safe and offsite storage space.

Regular backups are considered standard IT procedure, yet paranoid programmers like myself often feel it necessary to make their own backups just in case. At one small company, I was so distrustful of the backups (rightfully so - we eventually had a server crash just before a trade show and the backup tapes were unreadable), that I bought my own backup drives and media. But you don't want to rely on your staff to protect your data and in general you don't want them taking company data home (at least not just for backups). So do it right yourself.

Be Redundant

Management is often reluctant to spend a lot of money on hardware, especially given the inevitable obsolescence. But the cost of hardware pales in comparison to the salaries of programmers sitting idle because their workstation died and there's no immediate replacement, or the cost of a project delay due to a wait for critical equipment.

> Fortunately for me, in recent years I've been on projects where lack of hardware and software was not an issue. But the early part of my career was spent in large corporations and government projects where you had to jump through hoops to get the resources you needed to complete a project. I remember one military project where I was directed to order a new hard drive as "laboratory equipment" since we had more funds in that budget. The on-site naval representative responsible for vetting all such expenditures didn't seem to agree that was the correct categorization, but rather than immediately denying the request, he responded with requests for increasingly more information - what laboratory? what experiments will the hardware be involved in? submit a wiring diagram of the lab indicating how the hard drive will be connected!

You also want redundant knowledge.

> The first startup I joined that really felt like a startup had no dedicated IT person. My office space was a spot on the floor and we all set up our own computers. My manager was surprised when he saw me swapping tapes while the regular backup guy was on vacation, but really, neither skipping backups or denying vacations are good options.

> At another startup that wasn't quite as well funded, the air conditioning was off on weekends and evenings, including in the server room. It's a good thing I was shown how to restart the server - on summer weekends our IT person would call the office and say the server was apparently down, could someone reboot?

Make sure your IT person can go on vacation without the company grinding to a halt.

Have a Plan

Just as with software development, and for that matter, any other functional department of a business, there should be a long-term IT plan. Solely reacting to near-term needs will make transitions to future operations inconvenient, if not impractical. Think about how many users you will have to support, security and application needs, and future services that will have to be supported, and how much it will all cost.

> I was on a game project that had an excellent infrastructure consisting of cutting-edge third-party and custom development tools and a large, responsive IT staff. But just as things were getting busy and more development staff was added to get the project done on time, we got bogged down using the asset management tool - the number of licenses

> available were inadequate to keep everyone working simultaneously, and it took several weeks to acquire new licenses.

One tool that makes a difference between a run-of-the mill IT group and a top-notch group is a ticket system.

> I spent a good portion of my early career hanging out in IT offices and server rooms. Not just because I enjoyed their company and the air conditioning, but it seemed to be the best way to make sure they wouldn't forget my requests. When dealing with IT groups that employed ticket systems, in the worst case I could call up and refer to the ticket - in the best case, I had immediate responses over email and sometimes an immediate visit at my desk.

Know the Applications

It amazes me that IT often doesn't know the first thing about the applications it installs. Admittedly, I often can barely use the programs I develop, but still, I can at least launch them and invoke rudimentary operations. On the other hand, I've seen IT personnel unable to run the programs even to verify their successful installation.

> At one game development company where we had contract IT support, we had an ill-timed switch of our logins to new domain-based accounts over the weekend. Not only was this without advance notice, leaving people wondering on Monday why their original accounts were not functional (in particular, email, so there would have no point in notifying everyone by email after the fact), but many of the applications installed in the original accounts did not function in the new accounts.
>
> This fiasco occurred because the IT contractor was familiar with everyday-use applications like Microsoft Office and was apparently confident those packages were transitioned correctly, but was completely unaware that as software developers, we used quite a few other programs that were critical to our business, like compilers. And apparently the person managing our outsourced IT failed to consider that, too.
>
> The same IT contractor also went the extra mile to install virus-scanning software on all our machines and enabled them, again without telling anyone. And again, this is a good idea for normal computer usage, but many software code generation tools are documented not to play well with virus scanners.

Moreover, with dependencies and interactions (read, bugs) with different drivers, multimedia capabilities (e.g. versions of DirectX) and operating system versions and patches, it is important for IT to track issues with the critical applications used by developers.

Know the Business

IT procedures have to be in sync with the company's business. Practices adequate for maintaining regular 9-5 businesses may not be compatible with software development houses, Internet operations, or financial institutions.

> The aforementioned switch to domain-based accounts at a game development company not only went badly, but took place during the final crunch time stretch for that particular game. Remarkably, this happened again a year later at the same company with the same contractor - a new router was installed, once again without notifying anyone in advance, and worse yet, this was the morning of a business day, and once again during crunch time development. Considering that the employees were asked to put in extra hours in the evenings and weekends, this was really unforgivable.

When involved in packaging products, IT needs to know the application requirements and how to manage configuration changes.

> Another game company I worked at developed arcade games running on stock PC hardware. Our testers discovered a graphics glitch just as some new machines were being packaged for delivery overseas, and upon investigation it turned out that our hardware vendor had stopped offering our the graphics card used in our configuration, so our PC configuration people had started ordering a different card without considering that our game might not run correctly with it.

And in some cases, particularly web-based services, IT is on the front lines with the customer and should be acquainted with customer needs and expectations as much as anyone else in the company.

> One of my favorite projects was a wireless web browser and gateway for handheld devices. But for a consumer service that was supposed to run 24/7, we ran it haphazardly. After some heated discussions with my managers about adding features in before the launch date, I discovered our IT manager had already launched the service at his own initiative, with little fanfare. And when we moved our office upstairs for more space and a nice view of the San Francisco Bay, the gateway machines were moved in the late afternoon with no advance notice to customers or even staff. Any Friday afternoon commuters who wanted to browse the web on their train ride was out of luck.

It's not just small startup companies that get this wrong. I'm amazed in this era of Web 2.0 how badly prominent web sites are run.

> I was an avid review writer on Epinions until I lost several reviews-in-progress on Sunday afternoons - that is when when they scheduled their site maintenance update. Even Google showed some amateurish site

> management - I delayed setting up my Adsense account for a week because the password-retrieval page was down. The worst case I've seen of a site-that-should-know-better was the local Time-Warner cable broadband signup page. For at least a week the page stated it was down while an update was in progress, and for a while after that, it displayed the startup Apache server test page. If there was a truly competitive broadband market, imagine how much business they could have lost?

Know the Rules

Like HR, IT is part of every employee's tenure from day one. The rules don't just involve proper IT ticketing procedures. These days, every employee, and thus every IT practitioner should know the legal requirements and corporate policies governing privacy and proper use of the IT infrastructure.

> I worked for one delightful employer who went into muckraking mode whenever an employee left or was terminated - she would scour the former employee's hard drive and announce to everyone she found porn and lascivious email. Eureka!

Employees should know what expectations of privacy they have, who owns the data on the computer they're using, and what activities, e.g. porn-surfing, are restricted. (Although a friend of mine pointed out that some occupations are so thankless, Internet porn should be considered a job perk)

Management should also know, or be informed, if they're clueless, what lines they can't cross.

> In one of her more vindictive moods, the aforementioned employer floated the idea of breaking into a former employee's Yahoo webmail account, apparently assuming that was fair game if that account was accessed from work. It's not.

And you can't depend on the company legal department for expertise.

> While wrangling over a contract with a large video game company, I complained to their legal department that one of their clauses made no sense - it stated that any licensed components that I built into a deliverable would have to be sub-licensed by me to them without restriction. This indicated ignorance of how software is constructed (e.g. just building an installer typically incorporates installer code from the installer vendor) and how software licensing works (or even what the word "license" means). Ever read one of those interminable EULA's?

Communicate the Policy

Defining proper and improper employee behavior is always a tricky business, and computers in the workplace make it even more so.

> Terminating an employee is often a messy scene involving weeping, yelling, begging or all of the above. But sometimes what happens afterwards the employee leaves the premises for the last time is worse. People descend on that person's computer and discover porn, evidence of freelance work, and even sometimes root through email (one of my employers read an ex-coworker's email to his girlfriend)

You could argue that employees shouldn't make any personal use of office computers and anything they leave is fair game for employers who have to protect their interests. You could argue that it is in poor taste and unethical for employers to gratuitously root through all the leavings. Either way, make the rules clear - that's what employee handbooks are for.

You Can't Take It With You

The IT relationship with an employee doesn't end with his departure.

> One of my employers laid off several employees in a Friday afternoon massacre, notifying them at 5pm. One said he didn't have time to clear out his office and would have to return later to do that, including retrieving information from his computer. As this obviously was not a trust-filled working environment, the employer worried that this newly-disgruntled newly-ex employee might do some damage on the network. My suggestion: as a regular practice, as soon as an employee is terminated, back up that computer, save a snapshot of the final state on CD, DVD or whatever for easy access later, and take the computer out of service.

Taking snapshots of departing employees' hard drives is also a good practice for less cynical reasons. If the engineering team has to call up the employee and say "Hey, where is that documentation?" or "Did you forget to check in that last bug fix?" you're not out of luck.

I Miss Lisp

My favorite language is Lisp. What happened?

What I Miss About Lisp

Simplicity

A common complaint against Lisp is the parentheses. Lots of Interspersed Spurious Parentheses. That's pretty funny, but it's wrong. The parentheses are not spurious - each one serves a purpose, and if your program compiles, you have exactly the number you need. And there is no question about operator precedence or scope. I've never heard Lisp haters say they're confused by parentheses - they just find them annoying.

But a combination of curly braces, parentheses, brackets, periods and commas is simpler and more succinct? No one has ever had to pop open a Lisp book to remember its syntax (although I did wear out my CLtL2 referencing all the built-in functions). Think how many hours have been lost to Java and C/C++ operator precedence bugs, forgetting to add brackets when expanding a single-line if clause (and Eclipse will nag me about unnecessary brackets if I try to play it safe) and the endless debates among programmers about their favorite bracket usage and placement conventions.

Elegance

A related complaint is that Lisp is too strange and hard to understand. There is a misguided notion that programming languages should resemble English. Which has given us COBOL, BASIC and AppleScript. (I have fond memories of BASIC, and Microsoft had a pretty good run with it, but no one's going to write AI programs or even web server applications with it). English expression of computer programs is as efficient and clear as writing mathematical formulas in English.

Simplicity lends itself to elegance. You know you're using an elegant language if you're not distracted by discussions of proper style. I don't recall wasting time on coding style debates when working in Lisp. But after moving to Java and C++, nearly every organization I joined made a big deal about adherence to a coding standard, which usually involved some variation of the the abomination known as Hungarian notation. (Although Sun has defined a perfectly readable and simple style exemplified in the Java standard class libraries, and Hungarian notation is a monster when applied to object-oriented languages). A foolish consistency is the hobgoblin of little minds, the saying goes. So is a mandated coding style. When you're programming, you should be thinking more about the problem than the language.

Flexibility

The syntax makes possible a unique feature of Lisp, one that every satisfied Common Lisp user seems to appreciate - macros. "Macros" seems like such a mundane name - C macros or Excel macros are hardly in the same league. What other language allows you to implement new constructs that can even define new control flow, in a way that blends in with the built-in syntax? It's a small price to pay for parentheses.

After using Flavors, Object Lisp, C++ and Java, I still have to say the Common Lisp Object System is the best. Generic functions, eql parameters, multiple inheritance (yes, it's a good thing), a metaobject protocol that lets you customize the object system - nothing else matches it.

Optimization

A fallacy about Lisp is that it's slow. Of course, it's easy to write slow code in any language, and even easier in Lisp because 1) it's big, and tends to be slow just to start up, 2) automatic memory management and a run-time type system has an overhead and 3) Lisp is easy to learn - any idiot can learn it well enough to produce a complicated, messy prototype of something that looks sophisticated (although it takes an idiot with taste to appreciate it).

But it's really the best of both worlds. You can prototype an application quickly and clearly, and then optimize - at the algorithmic and data structure level as with any language, but also at the language level, e.g. by adding declarations that turn off run-time type checking. This avoids a scenario like the application I worked on that was prototyped in Tcl - and stayed in Tcl for two years while the development tried to figure out how to migrate it to another language.

It's painful to watch Java follow in Lisp's footsteps and yet apparently not learn all its lessons. I worked on a high-performance 3D content creation tool in Lisp - I still cannot imagine that a 3D Java application can perform as well.

Culture

Lisp had a cool culture. Much like AI labs, it attracted groupies of dubious ability but also some of the most brilliant programmers around. And I mean brilliant in a worldly, literate, sense. Lisp programmers don't make lists of design patterns, although Richard Gabriel did a lot of writing on how design patterns were originally applied to architecture. Among the well-known names involved in the origination, development and popularization of Lisp - Paul Graham, Peter Norvig, Guy Steele, Marvin Minsky, Rod Brooks, Richard Stallman, James Gosling....

Why Didn't Lisp Become Java?

A Hook

Whether it was a brilliant marketing move or stroke of luck, the introduction of Java initially for browser applets was crucial. It's not quite so popular for that, now (another essay - Why Didn't Java Become Flash?) but it was a great introduction that made Java an everyday word. Java hasn't exactly dominated the desktop and has been hit and miss in various areas, but has been targeted successfully in the web server and mobile phone arenas.

Common Lisp never had such a hook. It was associated with AI and large, expensive Lisp workstations. Now it is the first choice for, what?

It Wasn't Free

If you want everyone to adopt a standard, it has to be free. Netscape understood it and Microsoft understood it when they wanted to crush Netscape. Everyone who wanted to get into this cool thing called the World Wide Web was able to download Java and play around with it.

There are free implementations of Common Lisp (GCL, CMU Common Lisp, CLISP), but they came late and incomplete. The commercial vendors have been slow and late to release free versions of their products and still do only under restrictive licenses (evaluation/educational/ personal use). The open source versions don't have everything you really need, e.g. multithreading.

Class Libraries

Common Lisp is big, but doesn't include as a standard libraries that are important for real applications - multithreading, networking, user interface. Commercial Lisp implementations have their own versions of these libraries, but they're all different. For developers, the appealing part of Java's vaunted write-once-run-everywhere philosophy is not so much the bytecode portability - any polished app will have different installers for different platforms, anyway - but the fact that you can code to a single set of standard class libraries.

And Java has great class libraries. Not only do they supply just about everything you need and extend them as industry needs advance (without the lame Microsoft tactic of just adding new functions with "Ex" appended), they are typically well-designed - they express good API design, the standard Sun Java coding style, and model the underlying functionality well. For example, serious network programmers should read classic texts like Stevens, but just learning the Java network class libraries will give you a reasonable idea of what's really going on. And anyone concerned about coding style should just emulate conventions used in the Java API's.

Native Integration

Real applications also have to look like they belong on the machine they're running on. Java faced up to that in Swing - running with the Metal look and feel everywhere wasn't acceptable to anyone shipping a commercial product, so now you can switch to a Windows, Mac or Motif look and feel.

Too Big

It's easy to write "Hello, World" in Lisp. It just takes a while to load.

Too Good

The unique features of Common Lisp are largely unappreciated, even by Lisp devotees. I've seen macros used a lot, but mostly to inline code - few Lisp programmers think abstractly enough to define their own language within a language. I think CLOS is still the best object system out there, but most Lisp programmers I've seen are content to use lists. A West Coast Lisp programmer, during an interview for a startup using Dandelion isp machines, opined that "object-oriented programmers are born, not made". I'd have to agree - I've met many programmers complain about "too many objects", whether it's C programmers moving to C++ or Lisp programmers encountering CLOS.

In a sense, Lisp was too easy. A programmer who couldn't deal with C pointers could still hack together code in Lisp. I've seen plenty of prototypes that demoed well enough to get funding and turn into serious projects, then stall because the software was held together with string and bailing wire. And Lisp was too cool - it attracted a lot of groupies who weren't that great at coding but liked the culture and cachet (particularly in AI surroundings). So it's no surprise that Lispers come off as wine snobs to the beer-drinking masses out there.

Too Smart

It's a VB world out there. Most programmers don't really want to program - they want an IDE that will do as much work as possible for them, like graphically connect components and generate skeleton code. And they want the security of knowing they're using the same language, tools, API, design patterns, methodology...as everyone else.

Personally, I was happy running Lisp in Emacs, but if there was the equivalent of Eclipse for Lisp (or at least a VB-style environment), there might be a larger following.

Panic

While developing a commercial product in Lisp, I often heard, "No one will buy our product if they know it's written in Lisp", the implication being that we shouldn't tell any customers about Lisp, despite the obvious extensibility benefits, and that we should consider porting the whole thing to C++. For some reason, I never heard "We're having trouble selling the product because our salespeople aren't informed about the product, the company as a whole is uninformed about the target market, and we released the product with a lot of bugs."

I never heard any user complain about the implementation being in Lisp, but I did see one customer complaint about being "treated like a two-dollar whore" and I was called to a customer site to field angry tirades - I wasn't even on the project anymore at that point, but the regular sales support people couldn't take the abuse, anymore.

The attempts I've seen to move a project away from Lisp didn't solve anything and sometimes made things worse. Lucid doubled their size by buying a C++ company and then went bankrupt. A lesser-known CAD system that I worked on, called DROID and running on TI Explorer machines, had their own Lisp vs. C++ battle and somehow compromised on Smalltalk (now, what was the point of that?). Business people like to say that the product and its technology is secondary to smart business strategy and marketing, but when they screw up, it must be the technology.

Chess Moves: Avoid Getting Checkmated in Your Project

"[Chess is] as elaborate a waste of human intelligence as you can find outside of an advertising agency." - Raymond Chandler, in *The Long Goodbye*

I often hear and sometimes use sports analogies used for software development and project management (scrum, anyone?). But I think perhaps chess is a better match - I've often gaped at a project decision and thought to myself, anyone who played chess wouldn't have done that!

Look Ahead

One of my peeves - in television and film, the winner of a chess game always announces "Checkmate!" to the other player's surprise. In reality, that only happens when at least one player is so completely inexperienced or incompetent that they can't see the blindingly obvious. A chess game doesn't consist of two players bungling around at random until someone stumbles into a checkmate. Any strategy involves looking ahead at least a few moves, and any minimally decent player is capable of seeing immediate threats. Strong players can plan ahead several moves, and expert players can tell if a subtle weakness in position guarantees their eventual loss. In any case, chess players resign once they recognize the game is lost.

In fact, any decent chess program is based on planning - the further the program can look ahead, the stronger its performance.

It's amazing how often a project is started without anyone figuring out the steps to get there - the team stumbles ahead one move at a time until it's inescapably obvious that the project is late or infeasible.

Keep Cool

Chess isn't poker, but it pays to keep your composure. I played one opening so badly I lost my queen almost immediately, but my nonchalance seemed to convince my opponent that it was an intentional sacrifice, and eventually he crumbled. That was on the Iowa high school circuit, but I still as a high-schooler I entered a professional tournament where my adult opponent was so angrily surprised to find himself in a losing position that he just started moving my pieces to demonstrate he knew how I had the game won, then toppled his king. And I had no idea I was winning.

Manage Your Time

So you need to plan ahead, but you also need to balance that against managing your time. In tournament chess play, you have a fixed amount of time to play a certain number of moves. Thus your schedule is flexible, sort of. If you dawdle through the beginning of the game, you'll be pressed to move your pieces quickly later without much time to think about them. Experienced players are familiar with a standard set of opening moves and typically zip through those quickly and can allocate more minutes to the more complex positions later in the game.

I remember one chess game in which I squandered so much time, pondering one move that I had to move immediately, speed-chess style, for all the remaining moves. I managed to eke out a draw, but only because I knew exactly what moves to make without allowing any time for thought and any room for error. I've had quite a few software releases like that, too, where we dawdled, indecisively went down different paths, and in the end pulled mandatory all-nighters to hit a (in some cases immoveable) release date. In some cases, you could say the result was a draw. In other cases, our chess clock ran out. I remember working on one investor party demo even after the party had started - by the time we were ready to show it, everyone had finished their drinks and left.

Plan for the Worst

Another TV cliche - a person walks over to someone's in-progress chess game, takes a look and moves a piece, announcing "Checkmate!" Again, this never happens. Unlikely as it is for a player not to see himself losing in the next move, it is even more unlikely that the other player can't see the mate in one, either. It's like a prize fighter not seeing his opponent drop his guard.

Chess players plan their strategies assuming their opponents will make the best possible moves they can see. In fact, that's how chess programs are implemented.

In a project, you don't have to assume that the worst will happen, but you should plan for it. It's called risk management. You can't open yourself to the worst case and then forget about it. This is why you have insurance, this is why you don't blow your life savings in Vegas.

Do Your Research

A player at a tournament in Des Moines offered to show me his can't-lose opening. After I decimated him in a few moves, he said, wait, I've got another one. Again, a losing proposition. If there was such a thing as a guaranteed winning opening, the game would be no more interesting than tic-tac-toe. There's a reason for all those encyclopedic compendiums of chess openings and endings - a lot of chess experts over the years have tried them out and done the research, and there's no reason to try to relearn all that chess lore from scratch yourself by losing a lot of games for the next fifty years.

This is a common affliction in software engineering, too, sometimes referred to as the not-invented-here syndrome, but it's really worse - it's just making stuff up and thinking it works. Like the can't-lose chess opening, I've many a cobbled-up "algorithm" that just works for one or a few contrived cases and will break under real usage. There are decades worth of algorithms created in academia and industry, and research on complexity that can let you know what's practically solvable and what's not (calculating all the possible outcomes in a fifty-move chess game is an example of not). So there's no excuse not to start with prior work, and you won't embarrass yourself later.

Keep Your Ego in Check

I've been on a couple of high school chess teams where the person who thought he was best assigned himself the lead position, and then got skunked in tournament play. Once this occurred even after a holding an internal competition to sort out the rankings - I won, but the seniors couldn't believe that a sophomore was their best player, so they discarded the results.

This happens a lot in software, too. Everyone wants to be the CTO, software architect, or at the very least, lead programmer. But, obviously, not everyone is qualified.

Know Your Pieces

Chess players should recognize a relative valuation of their pieces: pawns the least valued, followed by knights and bishops, then rooks, the queen, and of course the king is critical. But more advanced players take in account the special capabilities and limitations of each piece. Pawns can only threaten two squares but can be extremely valuable when in position to queen. Bishops have range but are limited to either black or white squares - if you just have a bishop in your endgame, there's not much you can do. Knights can leap over obstructions but have limited mobility on the edges of the board ("knights on the rim are grim").

I've seen plenty of projects where people were placed in positions where they were ineffective - middle management positions without enough authority, technical lead positions without enough expertise...

The Sacrifice

I hesitate to bring up the chess sacrifice, but there's a pretty obvious analogy - burnout! In chess, overeagerness to sacrifice leaves you undermanned and losing. In crunch time projects, if the crunches arrive too early, not everyone is going to stick around 'til the end, or even stay effective if they do. And in chess, every game starts with a fresh set of pieces. Not necessarily so in software projects.

Lose Gracefully

You wouldn't think that chess is a violent game, but I've seen a few players angrily swipe all the pieces off the board after losing. Management likes to say, "Failure is not an option." Which sounds great when you're winning, but once you've lost, you have to decide how you're going to behave. Blame everyone in sight but yourself? Or give everyone credit for playing a good game, and move on to the next one.

If the outlook is grim or uncertain, you need to emanate confidence and competence. If you've recognized the game is lost, acknowledge it gracefully and move on to the next game.

Driving Lessons

> "Where we're going, we don't need roads." - *Back to the Future*

Backseat Driving

It's hard to avoid giving advice when you're sitting in the back seat. Or the passenger seat. It's also hard to avoid getting annoyed when you're on the receiving end.

> During my first job out of college, I often rode with several other coworkers who were not used to driving. The unlucky driver of the day would have an advisory committee monitoring his actions: "Slow down. Speed up! Watch out!" The constant micromanagement had the laudable intention of keeping the car and ourselves intact but, as one of the committee once remarked, "leave him alone, or we're definitely going to get in an accident".

Of course, a navigator can be useful. But only if the navigation is good.

> I once had a friend visit me in the Bay Area who wanted to visit a certain restaurant in San Francisco. He would say "Turn here!" just as I passed the desired exit. Not helpful.

Give directions plenty of time in advance. Even my Tom Tom says, for example "after one mile, turn right".

Check the Roadmap

You can minimize last-minute confusion by planning your route. An interesting type of phone call I occasionally get is when a friend calls me up and essentially uses me as MapQuest, but without the starting location. ("How do I get to A?" "Where are you?" "I don't know") This seems to happen mostly around LA, perhaps a commentary on the freeway system here, but here's an idea - how about looking at map before you start?

I'm embarrassed to admit I've succumbed to this overlay casual trip planning more than a few times, and my overconfidence ("I'll just figure it out as I go along" or "I'll ask someone along the way") invariably has resulted in delays and detours and sometimes nerve-racking moments through parts of town that look dicey. But when I check out the route beforehand, I feel in control all the way through and can usually get back on track easily if there's a minor diversion.

> Projects often start out the same way. We'll know where we're going when we get there. Then, months or years late, with the destination far on the horizon, management tells everyone to drive faster and and hires more drivers to help.

Figure out where you are and where you want to be at the end of your journey. Look at the obvious routes, and take into account the possible delays and detours.

Get in the Right Lane

Short-term planning is important, too. It's slow torture watching drivers cruise along in a lane clearly marked to merge into another, waiting until the last possible moment to merge. What's plan B - drive off the road?

> I've been on more than one project where it was suddenly announced "we're delivering a release today". When I explained to one manager that it wasn't a good idea to just tell everyone to stop coding and ship it, he explained, apparently sincerely, that we knew we were going to have a release so it should be no surprise. Another manager thought it was a show of leadership - "Today you will give me a build". Rudeness, on to of stupidity. Let's just say, those releases were not fit for final releases.

Don't defer decisions, whether out of complacency or indecision or a misguided desire to keep you options open, that have to be made. Execute them early, with a margin of safety and time for correction, and with everyone on board.

Use Your Turn Signal

If you do have to change direction, be clear about it. Among my many pet peeves are those drivers who are too lazy and inconsiderate to use their turn signals - there's a reason why those are built into cars at extra cost, namely to alert other drivers of your intentions so you don't end up driving into each other. And it is a courtesy - how often have you cursed drivers for cutting in front of you abruptly or leaving you waiting at an intersection for them to cross when in fact they were going to turn?

> It's not easy being a middle manager. I got stuck in one tragicomical situation where I wanted to move one recalcitrant employee out of a game design group - my boss agreed and yet within twenty-four hours assigned her a new game design task, after telling the IT guy to remove her from the game design mail list. The employee eventually got upset that the tasks she was unwilling to do were being assigned to others and started crying in front of me, jumping to the conclusion that she was about to be terminated. My boss then informed me she would be terminated, but apparently had a change of heart and I ended up hiding from that employee for a few more weeks while she gave me dirty looks.
>
> Now, if that was confusing for me, imagine what it was like for everyone else in the group who had to work with that employee. At least I was able to hide.

Unless you're working in the equivalent of Boston traffic, where signaling your intentions is a sign of weakness, then it is important to signal your project members whenever you're about to turn or make a lane change. If you can't give signal your intentions clearly, anyone with initiative will just give up and you'll be left with those who fatalistically and passively wait for micromanagement.

Are We There Yet?

It's an annoying question from kids in the back seat - it's just as annoying from your project manager.

Don't Drive Aggressively

I marvel every day in traffic when I see someone rushing past me to squeeze in front of my car when there's a hundred feet of clear road behind me. Some people just don't feel like they're making progress unless they're making it at someone else's expense.

> You can see that in the workplace, too. Despite all the platitudes about "win-win" solutions, the benchmark for others' success is often your failure. Some see life as a zero-sum game (though it's unlikely they are sophisticated enough to think about in consciously in those terms) Watch out for those sharks, and don't be one of them.

Drive Defensively

No matter how good your driving habits, you have to watch out for other drivers.

> The zig-zagging boss, the coworkers who cut you off.

Try to avoid the pile-ups.

Take a Whirl Tour on the Nintendo GameCube

These days the game industry is trying to follow "best" practices in mainstream software development, but one excellent practice that should be emulated from game development is the postmortem, as promulgated by Game Developer magazine. I tried my hand at it, submitting this postmortem to Game Developer back in 2003. I didn't get a reply, so the Nintendo Developer Support Group published it on their developer web site. Details covered by the developer non-disclosure agreement are omitted here (this was the version submitted to Game Develope).

Game Data

- Publisher: Crave Entertainment and Vivendi Universal
- Release Date: November 2002 (US) and March 2003 (Europe)
- Number of Developers: 15
- Number of Contractors: 2
- Length of Project: 15 months (plus 4 months for European SKU)
- Development Hardware: 1.5GHz PC's with GeForce 3 video cards running Windows 2000. GameCube devkits, test consoles and disc burners.
- Development Software: Microsoft Visual C++, SN Systems ProDG, Havok, 3D Studio Max, Adobe Photoshop, Adobe Premiere, CoolEdit, cvs.
- Project Size: 360,000 lines of source code. 900MB of compiled data.

Introduction

Papaya Studio's Whirl Tour was released on the Nintendo GameCube and Sony Playstation 2 platforms in November 2002 and March 2003, for North America and Europe, respectively. The game, published by Crave Entertainment and Vivendi Universal, combines a story-based adventure with Tony Hawk style extreme sports play on scooters. Although Whirl Tour is a multi-platform title and the merging of genres poses intriguing game design questions, this article focuses solely on the GameCube development.

We started work on the GameCube in July 2001, a few months after the Whirl Tour prototype was completed and running on a PC. Crave received milestone deliveries every two months - the first one involving a GameCube build took place in January 2002. We delivered a build to Nintendo of America (NOA) the following April for concept review, and Crave demoed the game at E3 in May. That summer, Crave forwarded our beta release as a presubmission build to NOA lot check, the group responsible for verifying compliance with Nintendo guidelines. The final submission was approved in September 2002 for the North American release. Over the next few months, under the aegis of Vivendi Universal, we undertook localization for the European release of Whirl Tour, which was submitted to Nintendo of Europe (NOE) and approved in January 2003.

What Went Right

1. Development Tools

When we started GameCube development, five months before the console showed up on retail shelves, development hardware was scarce and software tools were still under development. But we minimized the impact of this problem by getting started with Nintendo's software based emulator and obtaining the tools as early as possible and in sufficient quantities.

The GameCube emulator provides API-level emulation on a PC. Our team was already working with a PC reference implementation of the game using Visual C++, so while we waited for devkits (programmer version of GameCube hardware that contains twice the usual memory and loads data from the developer's PC instead of disc), we incorporated code that called the emulated video, graphics, disc and controller API's. In six weeks, we had a pseudo-GameCube version of our title running, albeit at 5fps.

We applied some optimizations to get the game running up to a slightly less painful 10fps, but there was always a nagging suspicion that time spent optimizing with emulator wouldn't result in anything useful on the real hardware. It was a relief when the first devkit arrived, two months after we started work with the emulator, but it took three months to really get the game built with the compiler (SN Systems ProDG) and running on the hardware. We had to wait for a GameCube port of the physics middleware, and we needed to update our offline data preprocessor to convert the little-endian data produced on the PC into big-endian data readable by the GameCube CPU (a custom PowerPC called Gekko).

Although we had to wait for a port of the physics middleware, it was still important that we received the devkits as soon as we did. Aside from the endian issue, there were a number of subtle differences between the emulator and actual hardware that had to be resolved, including differences between the Visual C++ and ProDG compilers. Likewise, we reaped benefits from obtaining the test consoles and disc-writer as soon as they became available. They arrived shortly before our first schedule GameCube milestone, so we were able to make that delivery to Crave on a test disc instead of a bunch of development files for a devkit.

Besides getting all the tools as early as possible, we obtained redundant quantities of each. The extra units saved time and provided peace of mind. For example, we ran 24-hour-plus smoke tests concurrently on multiple test consoles - when one console occasionally reported a fatal error and the others didn't, we concluded it was a hardware problem and didn't lose sleep over it (while were were losing sleep over other things). And when we had to return devkits for repairs (video card damage) and upgrades (PAL support), we could send back one unit at a time without stalling development. We also ordered hundreds of blank test discs in anticipation of daily burns and crunch time. The last thing we wanted was to miss a deadline for lack of discs.

2. Developer Support

We were fortunate in receiving thoroughly good support from all our licensors and vendors - Havok for our physics middleware, SN Systems for our compiler and IDE, and Nintendo Software Developer Support Group (SDSG) for just about everything else.

SDSG resources got us started on a lot of code. Our preliminary implementation of dynamic shadows, using shadow volumes, was guided by an SDSG white paper and refined based on ideas exchanged in the developer newsgroups. The final shadow implementation used shadow maps and was initially based on one of the SDK demos. For a screen snapshot function, we reused the source from a utility posted on the SDSG web site.

Access to SDSG was helpful throughout the project but proved especially so during the submission period. Our US submissions were rejected twice by Nintendo lot check, and our publishers wanted fast turnaround for each resubmission. Email and phone queries to SDSG resulted in quick clarification of guideline items and evaluation of proposed solutions. This enabled us to get the first resubmission out in less than a week and the second resubmission out in one day. Combined with lot check turnaround of a few days for each submission, the total time from first submission to final approval was just under a month.

The delay in getting a port of the physics middleware could have been serious, but a few weeks after we got our devkits, Havok sent us a linkable library that allowed us to complete a build. When it became apparent that we needed endian-conversion for our collision surfaces, they quickly sent us a run-time solution, and eventually a preprocessor solution that we integrated into our offline data converter. The GameCube port was basically complete in five months and we received incremental updates throughout the rest of the project.

We opted for SN Systems ProDG over Metrowerks Codewarrior because we were already using ProDG for the PS2. This minimized compiler differences between the two platforms (we already had differences in code acceptance between Visual C++ and ProDG), but meant starting out with a beta compiler. The few bugs we encountered were addressed quickly by SN Systems support, and the official compiler release was ready a few months before the first GameCube milestone. After that, patches and upgrades were readily available on the SN Systems support site, and they had quick responses to questions sent via email or posted to the gamecube.snsystems newsgroup on the SDSG site.

3. GameCube Architecture

Another reason we got the game running quickly on the GameCube is the friendliness of the architecture. The graphics hardware (Flipper) supports multiple dynamic lights, multitexturing, mipmapping, blending, and fog, so most of our basic renderer features were easy to implement (though optimization required some work - see What Went Wrong). The GameCube combines a unified memory architecture (UMA) with fast 1T-SRAM, allowing flexibility in managing textures and the video framebuffer.

The hardware features a generous 16MB of auxiliary RAM (ARAM). We used half of the ARAM for sound effects - given more time, we would have used the remaining ARAM to increase the fidelity of our sound samples. For songtracks, we used the hardware streaming direct from disc. The disc is small and fast, so we never had a problem with load times. The disc capacity is lower than a standard DVD, but at 1.5GB, it was plenty for our 900MB of data.

4. Memory Management

Although fast, the GameCube main memory (MRAM) is limited in size (24MB) compared to the PS2 (32MB). We stayed within the memory limitation by aggressively keeping on top of that issue. By running on the test consoles as soon as possible, we forced ourselves to reach the 24MB target early, within the first six months of the schedule. The UMA meant the video framebuffer occupied MRAM. Since PAL resolution is higher than NTSC, we sized the framebuffer for PAL in all builds to ensure we wouldn't have memory problems later. We configured all of the devkits to use only 24MB, so the programmers would know immediately if any level didn't fit. Testers went through all levels in each daily build and in their email report listed any levels that didn't load.

Flipper supports S3 texture compression, so we compressed as many textures as possible. Any such textures that didn't look good after compression were reworked. This considerably reduced memory usage (and as a side effect improved performance and load time) and didn't seem to degrade the visual quality noticeably. After E3, new assets pushed us over the memory limit, so we turned mipmap generation off until we had more space and a more selective procedure for mipmapping.

5. Testing

We made a point of testing early and often. Starting about a month before the first GameCube milestone, we released daily builds on test discs for our QA group. The builds were created from the latest code and data checked into cvs. Our testers issued a daily test report noting new bugs, verifying fixes, and listing minimum single-player and multi-player frame rates for each level (the build included an on-screen fps display).

Crave provided us with a tester from their staff who worked on-site with us and tested our daily build. At first, we sent our Friday build every week to the rest of their QA group, but in the last two months before submission we Fedex'ed them our builds, daily. We also set up remote access to their bug database and used that as our sole bug reporting and resolution mechanism. Since changes to the database would show up on the client GUI instantly, we would be alerted to the new bugs as they arrived and could investigate and update the database entry immediately with comments or requests for closure.

The Crave QA group provided valuable console testing expertise, including experience with a recent GameCube title, and were familiar with the Nintendo developer guidelines. Soon after they started testing Whirl Tour, their GameCube bug database contained about 200 entries, many of them memory card and disc error handling issues that would have been flagged by lot check at submission. By the end of the project, we had around 700 bugs logged and resolved for Whirl Tour GameCube.

6. Schedule

Originally, we estimated the GameCube development of Whirl Tour would trail the PS2 by one to three months, for reasons including the expected delay for hardware and middleware and less in-house familiarity with the newer GameCube platform versus the PS2. But that meant we would at least partially miss the critical holiday retail season. When it appeared that we could deliver our first GameCube milestone on disc, it made sense to synchronize the GameCube schedule with that of the PS2. This allowed us to target September for our North American submission, which would allow the game to arrive on the retail shelves in November. And it greatly simplified our overall project schedule, since we didn't have to track different milestones on each platform.

The decision to sync the schedules was consistent with our approach of top-down milestone-driven scheduling. Instead of building the schedule bottom-up by summing up low-level tasks, a method which accumulates error over time, we built the schedule top-down, placing the most importance on critical deadlines like E3 and the target submission date, followed by our milestone commitments to the publishers, and then juggling tasks to accommodate those dates. Adhering to the big-picture deadlines proved even more important later in the project, when Papaya was gearing up for the next project - any delay in this title would have just propagated to the next.

The schedule was tracked with Microsoft Project. Milestones were marked as such using the milestone task annotation in Project, and features required for each milestone were listed as subtasks, down to a granularity of no less than one week per task. We marked only real dependencies between tasks, no artificial links to force a nice-looking sequence, and we let Project calculate the expected dates. When a task was completed, we filled in the date for the finished task and let Project recalculate the dates for the remaining tasks. Thus we used the schedule not so much as a micro-management and task-scheduling tool than as a schedule feasibility tracker and early warning system.

We had weekly team meetings that initially involved only the leads and proceeded in a go-around-the-table-and-report fashion. Eventually, the format changed to include all team members and followed an announced agenda. This resulted in more focused and shorter meetings, sometimes lasting only fifteen or twenty minutes and rarely lasting more than an hour. Minutes were distributed by email afterwards. The meeting time was also minimized by requiring each team member to send a company-wide status report every Friday on his area of the project.

What Went Wrong

1. Performance

The biggest disappointment was our failure to achieve our original performance target of 60fps for all levels in single-player mode. We did achieve the desired 30fps for multiplayer mode in all levels, but for single-player mode we could only maintain a consistent 60fps in two-thirds of the levels. The remaining levels alternated between 30fps and 60fps, so we had to lock them to the lower frame rate to keep the visuals smooth.

We made a mistake in not achieving and maintaining the target frame rate early. We started with a consistent 30fps in our first GameCube milestone, and the code performance improved all the way through the project. We planned on achieving 30fps for the first milestone and reaching 60fps by E3. The fact that we only had a few levels running at 60fps by E3 should have been a wake-up call, but we succumbed to wishful thinking and hoped that further optimization techniques would get us to our goal. We did improve code performance by about 50 percent between E3 and submission, but with more NPC's, water effects, particle effects, dynamic shadows, and so on, we just kept pace with the additional assets. In the end, we still left a third of the levels below target in single-player mode.

It became apparent, too late, that many of our optimization techniques did not improve the worst-case performance in our problematic levels. If we had recognized this earlier, we may have been able to restructure the levels to exhibit more balanced performance. There were some architecture-specific optimizations that we did not fully explore, but again, we ran out of time.

2. Submission (and Resubmission)

The NOA submission was rejected twice before approval. That isn't a particularly bad showing, but there was a lot of pressure to get each resubmission out quickly, and it wasn't fun. We cut it close in terms of getting the release out in time for the holiday retail season, so it would have been more comfortable if we could have knocked it down to one or two submissions.

The most persistent problem areas that kept showing up on our lot check submission reports included memory card error handling and on-screen depictions of controller buttons. We could have gotten a pass on the button graphics, but we updated the button graphics several times anyway in hopes of making them more acceptable to lot check. The memory card issues for the most part were not negotiable - there are comprehensive guidelines involving error-handling flow and messages. These are not very exciting features to implement, but they are required, and we should have tried to get that stuff done earlier.

Our programmers and testers were not familiar with the GameCube submission issues, and as a result, we were overly reliant on Crave's QA team to identify the problems. Once Crave started testing the game, we basically had two months to rewrite our memory card, disc error, and reset-handling code, while simultaneously adding our shadow code, performance optimizations, and trying to fix all other bugs before submission. This was complicated by a couple of attempts in the last few weeks to advance the submission, which, in retrospect, now seems unrealistic.

We should have educated ourselves on the submission issues much earlier by reading up on the guidelines and examining other games, and we should have scheduled completion of all the guideline requirements by our beta release early summer. Crave sent the beta to lot check, but we didn't receive the results until two weeks before the final submission. The presubmission report from lot check was useful, but would have been more so if we had tracked it down earlier, and if didn't have so many violations (too many to fix in one pass).

We complicated the memory card situation by attempting to share much of the framework with the PS2. We delayed much of the GameCube-specific implementation until the PS2 code was nearly done, so most of the work was completed in the last three months before submission. The code sharing seemed convenient initially, but there were significant differences between memory card guidelines for the two platforms. The resulting error-checking and error-handling flow was somewhat atypical for a GameCube title, requiring clarification among the QA groups, lot check, and programmers. After hasty code mangling during the resubmissions to accomodate QA and lot check objections, the shared code was a mess. It seems obvious now that we should have completely separated the memory card code for the two platforms.

3. Manual Builds

Although we had a daily build, it was not automated. We performed the build on a developer station using the ProDG/Visual Studio environment. This allowed us to make quick fixes before releasing the build to QA, but it was not the type of clean and reproducible build that good configuration management practices mandate. An overnight batch build would have saved time (each build took anywhere from 30 minutes to three hours, depending on fixit attempts), been reproducible, and, if we set it up properly, automatically logged and emailed build results.

Moreover, an automatic build could have helped to establish a sense of regularity and process. The team members were never sure when the next build would be available, so eventually we had to send email every day announcing the build. Aside from the testers, most of our team members were still overly reliant on the PC version on the game and there was a slightly disruptive tendency to go straight to the programmers instead of checking the latest official build. And the fact that the build took place manually and not at any designated regular time made it too tempting to request new builds at increasing and irregular intervals, particularly during crunch times.

We had our data converter running every night by the end of the project, allowing us to check the latest art and sound effects changes. But the data build did not include processing of songtrack and FVM data, which were provided by external sources as WAV and AVI files requiring lengthy (hours to do a whole set) conversion to GameCube-specific compressed streaming formats. The FMV conversion was particularly involved, as we required versions for NTSC and PAL, and we had to extract the audio, resample and merge it back in to the converted FMV files. Unfortunately, the songtrack and FMV sources arrived fairly late in our schedule, so we were processing iterations of this data well past our attempted one-month data freeze before submission.

4. Wanted: Tool Programmer

We could have used a tool programmer. Our data converter was augmented by several different programmers in an ad-hoc manner, so new features were not consistently implemented with all the target platforms in mind. For example, although mipmapping and texture compression were technically multi-platform features, the code ended up in an "ngc-utils" file, and triangle-strip generation and texture palettization ended up in PS2-only code. A dedicated tool programmer could have developed the converter with more of a big-picture approach and taken some of the load off the other programmers, particularly the console programmers and lead programmer. A dedicated tool programmer could also have addressed the artists requests for previewing tools, which we just didn't have time to work on.

5. Lengthy Localization

The localization process for the European release took a lot longer than expected. Crave's QA group was regularly testing our PAL builds well before the NOA submission, so we expected to have a submission to NOE ready as soon as we had the text translations. The game engine modifications turned out to involve new texture fonts for the extended character set, additional character selections for text entry screens, resized GUI elements so longer European words would fit, and proper synchronization of the in-game language with the console ROM language setting. The code changes took a while, but a greater delay was incurred by an initial one-month wait for the first set of translations from Vivendi, then several iterations with Vivendi's QA group to fix translations, typos, and text trailing off screen. We made our submission to NOE three months after the NOA submission, and then it took an additional three weeks for NOE to reject the first submission and approve the second.

6. A Mild Case of Featuritis?

We didn't completely meet our performance targets, and we had a crunch time involving a flurry of fixes and resubmissions. So any work that didn't result in obviously necessary functionality warrants reevaluation:

Audio Effects

We had audio reverb and delay effects on our requirements checklist, but they were used in only one area in the game. It was gratifying to say we used the audio effects hardware on the GameCube, but it made our audio code layer significantly more complex and took several weeks to get it right. Compounding this, we were worried about the audio DSP dropping voices when oversubscribed, so after E3 we rewrote the system to handle this eventuality. But the rewrite took two to three weeks, the resulting code was messier, and once we had all the sounds in the game, the built-in GameCube metering hardware indicated that we were only using twenty five percent of the DSP cycles in the worst case.

Screen Capture

We had a publisher requested for a screen snapshot function, which we implemented shortly after the first GameCube milestone. It worked only on a devkit and was used by our testers for their bug reports. But it's unlikely that it was used by anyone else, internally or externally. Submission releases were not supposed to include debug code, so we ended up removing the screen snapshot code before the presubmission. All in all, we spent five months maintaining that feature.

Video Modes

Nintendo recommended but did not require support for progressive scan. We implemented it, but neither we nor our publisher's QA group had progressive scan monitors, so we relied on informal testing off-site using personal equipment. Our progressive scan support was noted by some reviewers, but there was a cost in coding, testing, and added risk to the submission, as Nintendo has guidelines on proper implementation of progressive scan.

Similarly, Nintendo recommended but did not require support for 60Hz mode in PAL releases. The number of users who can take advantage of this feature is probably higher than with progressive scan, and this feature was testable in-house and by the publisher's QA group, so it is a more justifiable feature But again, there was a development and QA cost, code complexity to handle both 50Hz and 60Hz modes, e.g. in selecting the appropriate FMV to play, and additional localization required for text displayed when switching modes.

Shadows - We implemented detailed dynamic shadows for player characters and enemy bosses. If we had limited the detailed shadows to just the player characters and used blob shadows for the bosses, at least two more of the levels would have run at 60fps instead of 30fps. It wouldn't have looked as good, but would have been no worse than many high-profile games that have recently appeared. We used higher-resolution shadow maps for the player characters, while allowing the boss shadows to have a more jagged appearance. The hi-res maps looked good, but the effect was subtle, and there may have been a performance penalty. Late in the project, we combined the shadow map blending with a location-based shadow value so the dynamic shadows would blend with pre-shadowed areas. Yet again, this looked good, but the effect was subtler still, and it was risky to make such a change right around submission.

7. IT

Our server crashed right around E3, and although we had an automatic tape backup running every night, it turned out the data was unrecoverable. Fortunately, we had local copies of the working data distributed around the office, and we had performed periodic manual backups on DVD. It took a while to piece together all our code and assets, and some of the work done just before E3 was lost. It could have been worse, but we did pay a the price for failing to ensure the integrity of our backup system.

After the server crash, we spent the next several months upgrading our IT infrastructure. This didn't distract too much from the ongoing development work, but we could have done a better job of minimizing disruptions by communicating and coordinating the IT plans. For example, switching from local accounts to domain-based accounts made some development tools unusable until some Windows registry values were updated. Even then, we were unable to upgrade our ProDG installation, which meant we were unable to upgrade the Nintendo SDK. And we had to use the original local account for our disk burns. We could have attempted to diagnose the problems or at worst perform a complete reinstallation, but with crunch time starting and the submission deadline looming, we really didn't want to take the time. This might have been avoided if we had discussed the IT planning in our weekly meetings and coordinated some upgrade tests and fallback policies.

Lessons Learned

This article focused on the GameCube development of Whirl Tour, but most of these issues are applicable to console development in general. As a console developer, many things are outside of your control, so control what you can. Get the development tools as soon as possible, and get more than enough. Make full use of developer support. Achieve memory and performance targets early and maintain them. Establish daily automated builds and testing early. Take care of the requirements before working on bonus features. Schedule around the really important milestones and then stick with the schedule. Do early and multiple presubmissions if possible. Plan for a few resubmissions in the end - allow a month. And localization will take longer than you think.

How I Met Unity

This is the preface I wrote for the first edition of my game programming book Learn Unity 4 for iOS Game Development. My preface was removed in the second edition, so here it is for posterity.

Technically, I first started programming on a TRS-80 in my junior high school library, but really I just typed in the same BASIC code listing from a magazine every day until the librarian mentioned I could save the program on a cassette. I'm embarrassed to recall my first reaction when I heard the library had a computer: "What's it good for?"

A year later, I saw the light when I got my hands on an Apple II. After cracking open the user manual and learned how to draw graphics in BASIC, I was hooked. Soon I was writing Reversi games (one in BASIC, one in 6502 assembly) and even a 3D wireframe display program.

In the intervening years I wandered the Windows wasteland and worked in small and large groups developing computer graphics and games. But fast forward to six years ago, when I happily got back into the Apple fold (now with Unix!) and attended my first Apple World Wide Developer Conference. Joachim Ante, one of the cofounders and the CTO of Unity Technologies, gave me an impromptu demo of Unity 1.5, and it was exactly what I'd been looking for — an inexpensive 3D game engine that ran on a Mac and able to target multiple platforms, including Windows, Mac and web browsers.

So I bought a Unity 1.5 Indie license as soon as I returned home (this was when Unity Indie wasn't free), later upgraded to Unity Pro, and was pleasantly surprised a couple of years later that Unity would support iOS! (I love it when a plan comes together)

In the meantime, my former employers at Hyper Entertainment granted me a license to port HyperBowl, a 3D arcade bowling game I worked on over ten years ago, to Unity and various platforms supported by Unity, so now I had a meaty project to work with, besides the smaller apps I'd been experimenting with.

It took me six months to get the first version of the HyperBowl remake running as a Unity webplayer, standalone Mac and PC executables and on the iPhone (and by the way, also Android, Linux and Flash). And really, it was three months actually spent using Unity if you subtract the time I spent figuring out how to extract the art and audio assets from the original game.

Over the next few years, HyperBowl and my other games got faster and better-looking with each new version of Unity and took advantage of more iOS features and more iOS devices like the iPad. I added capabilities using third-party plugins, a new pause menu, and even an entire new HyperBowl lane (level) with packages from the Unity Asset Store, which is conveniently integrated in the Unity Editor.

This has all taken place with a dev team of one (not counting all the work put into the original licensed assets), and I didn't have to learn a single line of Objective-C or create my own art! In a sense, I feel like I've returned to my programming roots, working on my own projects for fun, and as a bonus, profit! Hopefully, I can distill my experience with Unity over the last six years (both mistakes and successes) into this book.

About this Book

With any game development book there's the problem of trying to be all things to all people —there are plenty of areas in game development, Unity, and iOS that could easily take up whole books in themselves. This book is an introduction to developing games for iOS with Unity, so our goal is to get everyone acquainted with Unity and moving into Unity iOS development smoothly, following the same progression I've made over the years (but in less time!)

On the way, we'll go through step-by-step project examples to learn the Unity Editor and how to incorporate graphics, audio, physics and scripting to make games, and then we'll move on to developing specifically for iOS. We'll take advantage of free art and audio from the Unity Asset Store, so we won't be creating 3D models or sound samples from scratch and can stay within the comfy confines of Unity (except when we have to dabble in Xcode when making iOS builds). We will, however, do plenty of scripting with Unity's version of Javascript. Our focus will be on using the built-in Unity script functions, but we'll point you to Unity plugins and packages that provide further capability.

Explore Further

No one knows everything. That's why a key to successful development is knowing how to find the tools, assets, information and help you need. So instead of just listing a recap of topics at the end of each chapter (I never bother reading those), we'll suggest followup reading and resources for you to explore further.

This is a good place to recommend some other worthwhile Unity books. Even on the same topic, it's useful to read different books for their different takes on the subject. For example, Will Goldstone wrote one of the first Unity books, *Unity Game Development Essentials*, Sue Blackman's *Beginning Unity 3D Development* is a hefty tome that presents an adventure game, and Jeff Murray covers Unity iOS in *Game Development for iOS with Unity 3D*, using a kart racing game as an example.

Since I waxed nostalgic on Apple computers, I should follow up with a list of good Apple historical reading. *Revolution in the Valley* is a fun collection of Mac development anecdotes collected by Andy Hertzfeld. i*Woz* is an interesting peek at early Apple history and at the Woz himself.

While this book does use make heavy use of example game projects, we won't have much discussion on game design. But there's certainly a lot of interesting reading on the subject. My favorite game design book is Richard Rouse's *Game Design: Theory and Practice*, mostly a collection of interviews with famous game designers. And there's a bounty of game design articles and blogs on the web site Gamasutra (http://gamasutra.com/).

ACKNOWLEDGMENTS

Before we get started, I'd like to give a special mention to the people who helped me get started back in my Apple II days. My parents, who bought the Apple II that I ended up programming (and also a printer after they saw me typing out code listings on a typewriter — that junior high typing class really paid off!). My fellow Apple II programmers, Dave Lyons (now actually at Apple) and Cam Clarke (who made it to Silicon Valley but left us all too early), and Mr. Leaman, the computer programming teacher who let us hide out in the computer lab during pep rallies. That was time well spent.

HyperBowl on the Small Screen: A Postmortem

technicat

May 24, 2015

fugugames
24

Game Data

- Publisher: Technicat, LLC (Fugu Games)
- Release Date: July 2009 (App Store) and May 2011 (Google Play)
- Number of Developers: 1 (not counting all the original HyperBowl developers)
- Number of Contractors: 0
- Length of Project: 6 years (and counting)
- Development Hardware: A succession of MacBook Pros and iOS devices, Android devices, 1 PC.
- Development Software: Unity Android and iOS Pro. SVN, git, mercurial. GIMP. Visual Studio. FBX SDK. GarageBand, iMovie. Aquamacs.
- Project Size: Approximately 70,000 lines of C# and Javascript (some from plugins)

Introduction

After looking at the Game Developer/Gamasutra-style postmortem of a GameCube title I worked on back in 2002, it occurred to me it's about time I wrote another one. I've been holding off on a postmortem for my Unity version of HyperBowl, a 3D bowling game in which you bowl through various fantasy worlds (ancient Rome, the deck of a rocking ship, the streets of San Francisco…), until I achieved Flappy Bird success, but maybe I shouldn't hold my breath. I've been working on it for six years, now—I wouldn't go so far as to say it's a labor of love, but at least a work of mild fondness. So here's what happened.

What Went Right

Keeping Bridges Unburnt

This version of HyperBowl came about because I worked on the original HyperBowl, developed by Hyper Entertainment and first deployed as an attraction game at the Sony Metreon in San Francisco. I only worked at Hyper Entertainment for the first few months of 2001 but kept in touch with the Director of Development there, Aaron Pulkka (now running his own show at Rabbx).

Around the time Unity announced their upcoming iPhone support, I mentioned to Aaron that I thought that would be a good way to get HyperBowl running on the iPhone. He agreed and suggested I just ask the owners of Hyper, and whaddya know, they said sure, they're not actively developing it anymore, so why not. And thus they granted me a license to develop it for the web, Mac/PC, mobile and the Wii (basically everything that Unity supported at the time).

Using Unity

Unity was the right choice for the game engine. In a way, Unity chose HyperBowl, as I started using Unity in 2007 because it ran on a Mac and had a webplayer and I happened to meet Joachim Ante at WWDC. I only thought of porting HyperBowl to the iPhone a couple of years later when I heard the plans for Unity iPhone support and immediately thought how the screen aspect ratio of the iPhone was a good match for the tall projection screen of the original attraction game (nowadays, you can still find some of the attraction HyperBowls still running in venues like Dave and Busters with upended flat screen TVs replacing the original screens).

So using Unity was the expedient choice and default choice, but it turned out to be a good choice. I didn't realize at the time it would be as popular as it is today (I believe this was before they made the Indie version free) and this was before Unity Android. The multi-platform support is a huge advantage. Even though my focus is mobile, I've also made Mac, Windows, webplayer, Flash, and Mac widget versions of HyperBowl, and the only major code difference is the between the touchscreen and mouse-driven control code.

I could have eschewed middleware altogether, but I had enough trouble just resuscitating the code enough for the asset conversion, much less port it to multiple new platforms. And I would have had to implement or do without a lot of goodies that we take for granted in modern game engines, like bump map shaders and dynamic shadows.

Getting the Source Code

In addition to the license, Hyper Entertainment provided the original source assets, including the audio, textures, models and also the source code for both the arcade and Windows versions of the game. And not just the files, but the original development PCs, one with the arcade version and one with the Windows version, so I was able to compare them and choose one as a starting point.

The first thing I did after booting up the PCs, even before connecting to the Internet, was copy everything I could find onto DVDs and my own computers, then committing everything to version control. This was fortuitous, as I next made the mistake of connecting one PC to the Internet, and instead of something momentous like the Forbin Project, it bogged down downloading a gazillion Windows Updates and ultimately suffered a hard disk crash. The other PC eventually just crashed on its own.

Exporting to FBX

After a few unsuccessful attempts at running the original assets to Unity's accepted file formats through various converters (and enduring one vendor's "I am the super guru of 3D conversion software"), I decided I might as well export the data directly from the game and that Unity's preferred model format really was FBX. When I tried converting the game assets to OBJ format, for example, those files wouldn't import successfully into Unity, and then when I tried the OBJ converter included in the FBX SDK and got the same result, I figured Unity is just using those same converters to FBX under the hood, so might as well just skip the middleman.

So I got the old code to build just well enough, with a few generations newer Visual Studio and stubbed out calls to old middleware, to the point where I could launch the game and load the lanes (equivalent to levels). That's was good enough to integrate the FBX SDK and and have the game immediately write FBX files for each lane (level) on startup (and startup was about the only thing it could do).

In retrospect, this was by far the better approach, and not just because the other approach wasn't working. If I had been able to convert all the original model files, I would have had to manually import them into Unity and then manually reconstruct the scene hierarchy for each lane, or write some Unity Editor scripts to perform the reconstruction. With each lane as a single FBX file exported from the game, I just had a handful of FBX files to import and no piecing together with Unity was necessary.

Prototyping an App

While I was trying to figure out how to convert the original HyperBowl assets, in order to test my idea for the bowling controls, I created a no frills bowling app, called Fugu Bowl (in the mode of Ford cars, I preface my other app names with Fugu, under the Fugu Games label. It's a long story that involves watching the Food Network late at night).

The control I had in mind, swiping the screen to roll the ball (although with Unity's game engine, PhysX, it works better to push the ball rather than apply a torque), is intended to emulate the trackball-like operation of the original bowling ball controller. In fact, the arcade version essentially had a trackball, whereas the original attraction game featured a real bowling ball that required you to put some weight into it.

I thought the result worked pretty well, and the app was surprisingly popular, despite consisting only of a monochromatic ball, capsules for pins, a plane for the bowling lane, and no bowling game rules beyond playing kids boos and cheers for gutters, spares and strikes. This was probably largely due to introducing the app in the early days of the App Store. It was my first app (another reason to release a simple app, first — figure out all those App Store submission intricacies), but it does pain me a little bit that my first app is still my most downloaded app.

Fans on Facebook

Of course I did the social media thing: twitter, google+, youtube…but the most successful platform has been the HyperBowl Facebook group. Currently it has a bit over 800 "fans", but more important than the quantity, some of them are loyal, supportive, enthusiastic fans who provide suggestions, requests, bug reports, and even localization (got some help on Chinese and French). Showing the benefit of a licensed title, many of the Facebook fans enjoyed the original attraction or Windows version of the game.

Quick tip: you can set up your Facebook page to automatically share posts to a twitter feed. This hopefully will new users to both your game and the Facebook page and saves you the trouble of manually posting the same news on two different social platforms.

Indie Is In

The first reviews, among all my apps, included a lot of negative reviews. And by negative, I mean snarky, condescending and amazingly prone to affront by a cheap or free app. That's pretty rare, now, and I don't think it's because my apps improved. The biggest difference, I believe, is that indie is in. People sympathize with the single developer or small studio and recognize there are limited resources, and even if they don't like the product, they appreciate the effort. Even now, a harsh criticism can turn into a supportive message if I respond, hey, I'm the only one working on this.

That's from the peanut gallery. The other source of snobbery is from the I'm-a-gamedev-pro-and-this-isn't-AAA crowd. I've seen comments like this in gamedev forums, too, but when that game designer who's used to working in a studio has to come crawling back for simple help in scripting, the attitude changes. By now, enough game developers have gone from salaried work to budding iPhone game developers (and then often back to salaried work — there really should be a rotational system where we can swap places), AAA elitism seems to have dissipated.

What Went Wrong

Exporting to FBX

Converting the original game assets to FBX was definitely the way to go, but it wasn't fun. When people say they dream of becoming game programmers (people do say that, right?) I'm sure they're not talking about becoming toolchain programmers.

Amazingly, I could not recognize a single line of HyperBowl code I wrote that long ago, even though there are portions I'm sure I must have written (I was not one of the original HyperBowl programmers, but I put in some optimizations and updates to work with newer graphics hardware). But whoever wrote it, that code was generally clean — it's not hard to traverse the scene and gather all the model and texture info.

The part I didn't like was working with the FBX SDK. Contrary to popular belief, FBX is not a file format. Sure, it has a file format, but it's deliberately undocumented — they tell you not to mess with trying to read/write the file yourself, but instead do everything through the SDK, which changes whenever (when FBX is called an "industry standard", that just means Autodesk, kind of like the Borg was industry standard). So I had to make sure I was writing a version of the FBX mystery meat that Unity could read, since it was running on the previous version of the SDK, and that involved some weird string matching in the writer initialization (as I recall, there's no way to say "I just want that one").

The irony is that the game files I was converting to FBX were actually in a proprietary, but obsolete, format that had the same model as FBX — undocumented format, always use the SDK. Because the vendor will always be around, right?

Losing the Source Code

Not to fear, I didn't lose the source code! At least, not the original source code. And I've got all the Unity projects also hosted offset in version control. No, the stuff I lost was the jury-rigged FBX export I wedged into the original game. I probably wasn't careful about saving it because I figured I'd never need to run it again. In fact, I was so sick of repeatedly exporting to FBX, importing into Unity, realizing something was off then tweaking and repeating the export, that I left some details up to manual fixes inside Unity.

For example, the texture coordinates were coming out flipped from the way Unity was interpreting them so all the textures were displayed upside down. That's something I should have fixed in the exporter, but I had reached the point where I couldn't stand it anymore and just manually adjusted them within the Unity Editor. But that means I'd have to do it every time I reexported. However, one PC crash and a few years later, when I thought about adding some optimizations to the export (e.g. the camera is always facing one direction as you bowl, so every static backfacing poly in the scene could be removed) I was faced with the prospect of reimplementing the whole thing. So that's it, as long as Unity can read FBX files from 2007, the pipeline is closed.

Too Little Marketing

I think the HyperBowl Facebook page looks good and has a decent number of followers, and I'm doing OK in general on twitter (though that's not HyperBowl-specific). But my other attempts at marketing have been at best sporadic. Theoretically, I'm willing to put spend on marketing when I have the time and income, but that's like saying I'll do it when I'm unemployed yet making money.

When I'm between paying gigs, I do at least put more effort into marketing, and I think it does make a difference in sales. Putting out promo codes, frequent updates (which is a way of getting more promo codes), Facebook posts, localization (China sales are up), contacting reviewers and bloggers, that all seems to help, although with the market getting more and more crowded, everything makes less and less of a difference. To be honest, my all-time revenue graph is more or less bell shaped, with the peak occurring a couple of Christmases ago.

A note about piracy. For a while, someone was cracking my iOS app within hours of every update. I find that obnoxious, and contrary to what some piracy defenders say, it didn't seem to help sales. But it didn't seem to hurt sales, either. On the other hand, in the past few months the number of cracked Android APKs have proliferated (not just download sites but also activations, dutifully tracked by Unity Analytics), and during the same period my Google Play sales have dropped dramatically. Probably unrelated, but hmm…

Too Little Monetization

When I started making apps, I didn't really consider incorporating ads, partly because there wasn't a convenient way to try out an ad service without writing your own Unity plugin, and partly because I don't like seeing ads, myself. So for every app, I charged or left it free. But one day I saw that prime31 had introduced an iAd plugin, so I tried it out on my trustworthy testbed app, Fugu Bowl, just so I would know how to set up iAds. Several months later, I checked my ad revenue and realized I was making some money. "Ads for everyone!" I proclaimed.

Unity now includes API access to iAds, so the prime31 plugin is not necessary (I keep it around as a fallback just in case — for a while, the early Unity iAd implementation was not as polished as the prime31 plugin). The generalized Unity ad API doesn't include AdMob or anything else on Android, so there I'm still using a prime31 plugin. I might try Unity Ads at some point, or Chartboost (I've received some developer recommendations), or maybe I'll drag myself onto the freemium bandwagon and implement in-app purchases. ("IAP for everyone" I will proclaim)

Too Much Monetization

By too much monetization, I don't mean too much money. I wish. Rather, one of my few real regrets is wasting time trying out various mobile ad networks just because someone emailed me. First of all, it's just annoying that the account rep always want you to set up a skype chat instead of just providing a documented SDK. And then it's laborious explaining to the rep later, slowly, that the reason you're no longer using it is that it earns a pittance or it doesn't work the way you want (or at all).

I realize I may not be getting the best revenue from my current ad networks (iAd and AdMob) but my current setup is convenient and reliable, and the time spent switching back and forth between other ad networks and plugins is better spent trying to attain Flappy Bird numbers.

Too Little

This may seem like just a technical detail, but after importing all the lanes, I discovered all of the geometry was smaller than expected by a factor of 100, assuming one distance unit in Unity should represent one meter (and it does, if you go by the default physics settings, lighting distances, camera scripts and so on). At first I went with my usual theory, that the artists screwed up the original source assets, and I went about adjusting the physics parameters, lighting and culling distances, camera scripts and so on. After all, theoretically it's unitless and arbitrary, so this should work. What could go wrong?

Well, physics went wrong. The game was playable and looked right, the pins fell at more or less the right speed, but they didn't always fall — sometimes they just kind of tilted. There was other weird stuff, all probably due to numerical precision issues. Or maybe I just did something wrong. But finally, I realized that Unity has an import scale factor that defaults to 0.01 so that it just works with the conventional use of centimeters in Maya, and when I reimported everything with no scaling and set all the other values back the way they were, Earth physics was back (I always wanted to add a moon gravity lane, though).

Lessons Learned

This just goes to show that you can learn a lot about a game engine with small projects, and that's probably the best way to start, but to really learn a game engine (and game development, for that matter), you need to create a large project that exercises a lot of features all together. In fact, I was invited by Apress a couple of years ago to write a Unity iOS book, and my first inclination was to make a whole bunch of small examples that demonstrated different features (touch input, accelerometer…) But trying to plan out the structure of the book, I ended up making another simple bowling game (sort of a Fugu Bowl with HyperBowl aspirations) so you can see how it all comes together.

The cross-platform support of Unity is still paying off. And I'm still paying for it — those Pro licenses for each platform does add up — but if I had tried a straight port of the original

HyperBowl code, I'd probably still be just working on the iOS version. And while I didn't get sidetracked into Blackberry or Windows Phone, a Wii U version may be on the horizon this year…stay tuned.