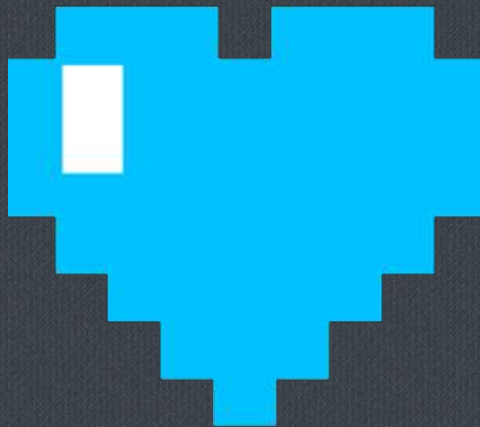

Videogame Developer's Strategy Guide



By Chris DeLeon

Edited by Laura Schluckebier

CHAPTERS OVERVIEW

	Why a Strategy Guide?.....	iii
1.	Getting Started.....	I3
2.	Questions About Education...73	
3.	Programming.....	IO9
4.	Get Motivated.....	I70
5.	Game Design.....	208
6.	Level Creation.....	265
7.	Team Projects.....	299
8.	Industry.....	348
9.	Game Analysis.....	405
	Image Credits.....	443

WHY A STRATEGY GUIDE?

Learning to develop videogames is itself an open-world adventure. Explore in any direction you'd like. Develop character skills that are broad, or specialized. Build a party, or go solo.

While this book can be read cover-to-cover - I've sequenced these chapters and sections to help that go smoothly - for most people it will work much better if used as a strategy guide.

If, in playing a game, you ran into trouble in the Lava World, or wished to learn about Enchanted Crafting, you'd head to that section of a guide. Similarly, here, if you're tinkering on a game type that has levels, jump to the Level Creation chapter for terms and processes. If you're in a group, browse some sections on Team Projects.

There's also one more way that this is like a strategy guide: tips in a guide only help if you're actually playing the game it's for. This guide is for videogame development. For these ideas to have context and purpose, you really should be actively making videogames of your own.

If you haven't yet started creating games: today's a great day to finally begin your journey!

CHAPTER 1

Getting Started

1.1	Making Your Own Videogames at Home is Totally Awesome	14
1.2	How Long Does it Take to Learn Game Programming?	19
1.3	Hobby Game Development: 20 Questions	25
1.4	Beginners Shouldn't Start with a Design Document	39
1.5	Clone Videogames to Learn Real-Time Videogame Design	42
1.6	General Concepts for Beginning Developers	45
1.7	Learn Videogame Development Like Woodworking	65
1.8	Fan Habits and Focus are Not Developer Habits and Focus	70

CHAPTER 2

Questions About Education

2.1 Advice to a New Student in Videogame Design	74
2.2 Questions from an Elementary School Class	78
2.3 Class Questions About Game Development Career	83
2.4 Math for Videogame Making (Or: Will I Use Calculus?)	89
2.5 Question About Comp Sci. and Game Development	94
2.6 The Ways of Self-Education	103

CHAPTER 3

Programming

3.1 Game Programming Fundamentals	110
3.2 How Programmers Program	124
3.3 Position and Speed Variables	132
3.4 Float and Int Variables: Casting and Other Issues	138
3.5 Hack Then Refactor	145
3.6 Basic Real-Time Videogame Artificial intelligence	152
3.7 Quick and Dirty Back-Ups	160
3.8 Steps in Programming a Simple Realtime Strategy Game	165

CHAPTER 4

Get Motivated

4.1 Stop Trying to Learn Everything Before Starting	171
4.2 "Overcomplicating Everything"	176
4.3 Think by Building, Build to Answer Questions	184
4.4 Your Attitude Matters Even When Working Alone	187
4.5 Don't Wait for an Event, Job, Contest, or Assignment	191
4.6 The Brain is Not an Emulator	193
4.7 Stop Arguing About What Makes a Better Game	200
4.8 Start Before You Have an Idea	204

CHAPTER 5

Game Design

5.1	Modest First Projects and Incremental Learning	209
5.2	Bottom-Up vs Top-Down Game Design	224
5.3	Photographer's Algorithm	232
5.4	A Little Planning Can Go a Long Way	236
5.5	Doing More With Less: Short Videogame Design	242
5.6	Colorful Oceans and Chunky Sauces	251
5.7	Genres and Conventions: Known Patterns of What Works	257

CHAPTER 6

Level Creation

6.1 Anti-Design / Backwards Game Design in GoldenEye	266
6.2 Level Design Concepts	273
6.3 Level Design Process	281
6.4 Level Design Q & A	288
6.5 Non-Essential Level Art is Essential	293
6.6 Visual Language in Super Mario Bros Level Endings	296

CHAPTER 7

Team Projects

7.1	Videogame Project Management for Hobbyists and Students	300
7.2	Communication is a Game Development Skill, Part 1	315
7.3	Communication is a Game Development Skill, Part 2	322
7.4	Establishing a Videogame Development Club	337

CHAPTER 8

Industry

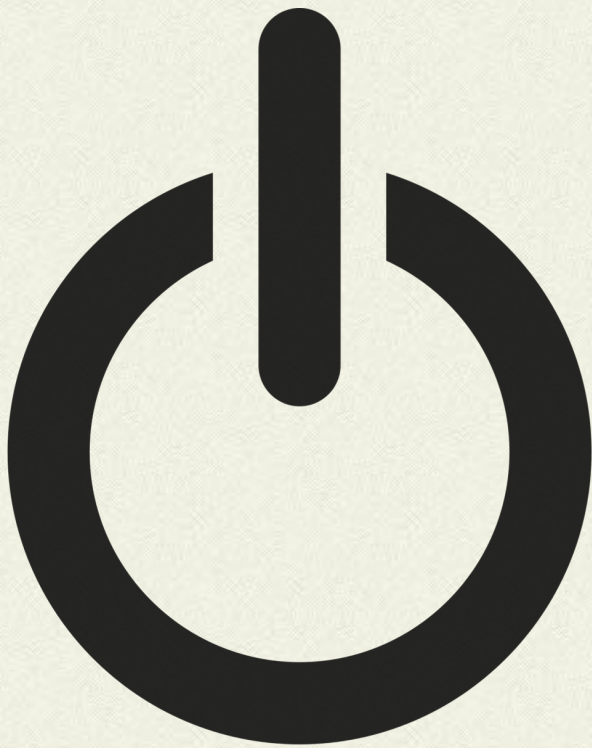
8.1 A Frank Look at Making Videogames Professionally	349
8.2 Indie Game Development as a Career	358
8.3 Influence of Business Models on Game Design	368
8.4 What's Japan Doing Differently	375
8.5 Intellectual Property and Hobby Game Development	380
8.6 Should I Release a Game as Soon as It's Done?	392
8.7 How to Get the Most Out of Your First GDC	396

CHAPTER 9

Game Analysis

9.1	Why Minecraft Worked in 2011	406
9.2	Reflections on Three Specific Hobby Game Projects	422
9.3	Quit Smoking: Effect of Interaction on Interpretation	434

1



If this is the first material that you're reading about videogame development: welcome! The way forward is long, difficult, at times frustrating... and totally worth it. You're on your way to learning how to create your own videogames! There's nothing better than homemade fun.

GETTING STARTED

MAKING YOUR OWN VIDEOGAMES AT HOME IS TOTALLY AWESOME

There has never been a better time than now to design and develop your own videogames. A typical home computer is all that's required. The ways to learn and share your games are already huge - and growing!



The original *Pac-Man* took a team of nine professional engineers a year to create. Although we tend to experience *Pac-Man* today as mere software, either downloaded or played in a web browser, the original was a 400 lbs. (180 kg.) painted wooden arcade cabinet of custom electronics that needed to be shipped all over the world.

Pac-Man's development and distribution required a level of labor and equipment in 1979 that could not have been achieved as a part-time side project.

Now in 2014 virtually any decently experienced amateur can remake a videogame similar to *Pac-Man*, working alone, mostly in an evening. It doesn't even cost anything to make.

Then you can distribute the game worldwide instantly for free.

What at one time revolutionized the game industry is now less complex than projects that we see made in weekend game jams.

Things got better. Much better.

BETTER FOR ALL OTHER GAME MAKERS, TOO

Advancements in development and distribution options means that we're seeing more kinds of games, released more frequently, from a larger number of creators.

This is very exciting, for players and game makers alike. But it's essential to recognize that this does not mean that making games *as a business* got easier, too. It's true that it's easier to sell an app on a smartphone than it was to sell a game for the old Nintendo Entertainment System or Xbox 360, but that truth applies to every developer in the world. In August 2013 there were multiple times more new mobile apps published *per day* (~2400; [VentureBeat](#)) than the total unique games released ever for NES (~800; [Wikipedia](#)), or Xbox 360 (~1,139; [Wikipedia](#)).

The barriers to entry to making games aren't just lower for you. They are lower for millions of other people making games, too. *If your main goal is to make indie games professionally, know that you will*

be competing against more people than ever to do so.

I'm not saying this to discourage anyone. I'm saying it to establish perspective of what to prepare for.

DANGEROUSLY MISLEADING MYTHS

There are some misleading stories out there which have tricked people into thinking that making videogames is a get-rich-quick scheme, or that quality and hard work aren't necessary to succeed.

The first misleading myth is that of the overnight millionaires. Some indies have fared very well. They are talented, and they put in the work, but their success appears to be overnight because you usually haven't heard of the first 30-50 games they've worked on, or seen what went into developing their hit for years leading up to launch. Most indie developers make very little money from their games, and many do paid contract work on the side to make ends meet.

The second misleading myth is that simple games that anyone

can make easily often explode into meme-like fame. The few games like Flappy Bird are the exception, not the rule. For every tiny, simple game you see that becomes a hit, tens of thousands of little games quietly flopped.

Tossing tiny, unsophisticated games into digital stores just in case one takes off is like buying time-consuming lottery tickets.

The vast majority of successful games were not simple projects or lucky breaks, but the result of hard work, talent, and experience. Nor does every game that arises from hard work, talent, and experience become a hit. No one gets any guaranteed profits or benefits from effort or seniority. Every game is a new, separate effort to excite and engage demanding audiences.

To make games professionally your work and skills need to be extraordinary. It'll require practice, persistence, experimentation, compromises, and humility along the way.

MOST CUSTOMERS AREN'T GOOD TEACHERS

The marketplace can be a brutal, risky, and unforgiving environment, which doesn't make it very friendly to beginner learning. People that insist on releasing their first games commercially just to see, and "maybe earn a little on the side" are seeing discouragement from dozens of dollars or less for months of their work. Would we really expect otherwise though, if someone just learning guitar sold their practice attempts on iTunes?

For this reason I advise people who are in it for the long-haul to at least begin by making games non-commercially. What can be done now without funding is incredible. It's a great way to start without betting the farm, nor mistaking a lack of paying customer response as a reliable gauge on whether you're learning and progressing.

Like a person going to Hollywood aspiring to be a movie star, if the goal is to earn a living making games as an independent developer, it would be prudent to

have a backup plan, a day job, and an understanding that it'll be a long, bumpy road to success.

BUT IF WHAT YOU WANT IS TO MAKE GAMES

If your near-term goal is to make games, rather than to make games as a business, then this is without a doubt the best, easiest, most exciting time yet to be doing it.

You have so many options of ways to learn, ways to create, and ways to share. I've actually heard this major benefit turned around and used by someone as an excuse: there are so many ways to make games now that they "can't figure out the best way to start." Absurd! There is no one best way to start, nor one best way to make games.

Anything that gets you making games and trying out your ideas is great. Within this book I'm going to give you the very best advice and ideas that I have to offer, but part of my advice is this: learn from any and every source that you can. I'm but one of many people who have been making games and is excited to help

others. Not sure whether something is worth learning? Just learn it. Then you'll know if it was.

Along your way you will learn and use many different programming languages, tools, and changing development environments. Don't get stubbornly stuck on whichever you use first, and then it won't make much difference which it is.

Know too that if you want to make videogames, learning can't just be book smarts. Your attention has also be on *doing* a practical skill. Reading can be helpful, a good use of time, but reading alone is not enough, for the same reason that someone won't become a great painter without a canvas.

GET LOST IN FOLLOWING YOUR INTERESTS

If you wish to establish an identity of your own as a developer you can't get caught up in chasing trends. Find and develop your own voice by following your passion.

"Not amusement nor distraction, but the desire to effect some cherished purpose is the strongest motive that can move the learner."

-John William Adamson

In order to create your own little worlds, you've got to be ready to live a bit in your own little world.

Well before stressing about what the public thinks, perhaps focus first and foremost on impressing and surprising yourself with just what you can accomplish.

Pull yourself from distractions, zero in on developing the skills that you wish you have. No one can create your many games the way you want them but you.

Your games will exist. Make up your mind to figure out whatever you must as you go to make it so. Accept that your games existing in an imperfect way is better than them not existing at all. (Anything that ever exists is imperfect!)

ENJOY THE FREEDOM

We're so fortunate to be alive at a time when you no longer need anyone else's money or approval to make your game a reality, nor to get your game in front of other players around the world. Making an enjoyable videogame no longer

requires special gear or privileged access to retail distribution. You can learn to do it in your free time, free from external pressures, and free to share however you wish.

You can bring new games and ideas into existence simply because you want them to exist.

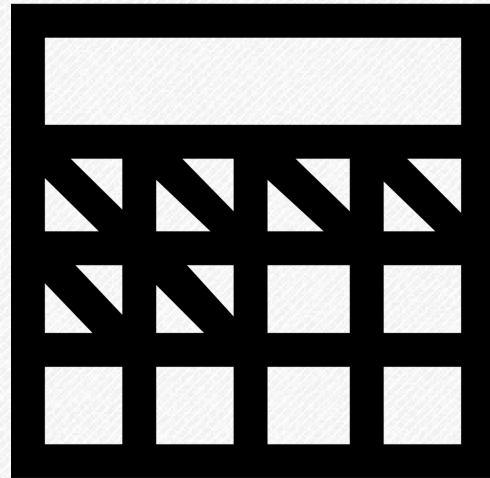
If there turns out to be a smaller audience for it – whether a circle of your friends, a dozen people across the globe, or even just you as the creator – that doesn't much matter. A game developed for free doesn't have to sell 100,000+ copies just to justify its existence.

As developers we've never had this level of freedom.

It's a great time to be making videogames.

HOW LONG DOES IT TAKE TO LEARN GAME PROGRAMMING?

Some people think they can learn game development in a weekend. Others think it has to be a career decision. Both are partly correct. Like a good game, videogame making can be easy to start but is never truly mastered.



A friend asked this question the other day. To peers nearby with more programming experience than him, it seemed like a silly, almost nonsensical question. Whether or not that's true, explaining why someone would think that sheds light on qualities of programming that may be invisible to someone newer to it. It's a useful question either way.

Game programming is not something someone goes from not knowing to knowing, in the same way that someone goes

from not knowing how to drive to knowing how to drive, or from not knowing how to read to being able to read. Even disregarding other purposes of programming, looking specifically at programming for videogame creation, there is an enormous range of potential outcomes, and many different metrics by which someone's work might be evaluated.

The question is a bit like asking how long it takes to learn how to paint a picture, or how to play an instrument. Getting set up and

putting a paint-covered brush on canvas or producing a musical note can probably happen on day one, but after that there are people who devote decades of their lives to becoming better and/or more versatile at these tasks. Then there's everyone in-between, from people a few weeks or months in (able to poorly mimic the prior work of others), to those a few years in (able to effectly mimic and perhaps personalize the prior work of others), to those several or more years in that begin to identify with the art, striving to excel in either classical execution or exploring to discover a distinctly personal style.

Game programming is similar. Getting a development environment installed and configured to turn sample code from a book or website into a runnable application can probably happen on day 1, or at least within the first few days depending on the amount of troubleshooting needed. Once that works, there

are a few fundamental concepts and new vocabulary words to pick up, then a bit of puzzling over example code to make sense out of why it behaves as it does. This is typically followed by being able to modify example code to achieve slightly different results, increasingly so until someone's familiar with enough patterns and basics to start building something on their own. Even still, that first something will usually be quite familiar to previous dabbling, more a partial mash-up of ideas than a project made from scratch, but there's nothing wrong with doing that to gain practice.

From here, there are countless directions in which to grow in the art of game programming. Someone might focus their energy on writing well-documented, easily understood code, learning about best practices to work more effectively with one or more other programmers on a team. Such skills are essential for career programmers on large teams,

although their importance among solo and hobby developers varies so long as they are able to produce the results they desire. Someone could become a specialist in highly technical details or specific problems that other programmers count on libraries or engines to solve, learning how to write better network code, graphics rendering/effects, memory management, custom file formats (such as levels, packed images, etc.), or ways to address common issues involved in making in-game artificial intelligence. It's also an option within the repertoire of a game programmer to delve into hardware interfacing and the construction of custom input or output devices, to make games utilizing new types of controllers (think about how games like DDR and Guitar Hero might have started) or unusual forms of output (tactile feedback, external LEDs, etc.).

Then there is the matter of learning more programming languages and development tools. Occasional shifts in technology – along with an acquired sense for when it's appropriate to switch between competing technologies – can feel like setbacks at the time, but in the long haul they can save a lot of time and effort if done wisely.

Also like painting or playing an instrument, some people may find that it comes more naturally, leading them to produce more presentable efforts earlier in their exploration, while others may need a bit more time to fiddle and experiment before feeling as comfortable. How long someone takes to feel like they have a handle on it isn't necessarily a sign of how much they'll accomplish with it though – like any creative endeavor, persistence and plenty of practice will gradually outpace others that don't work hard at it.

Any game programmer is also able to stretch themselves too thin,

taking on a task too far past the edge of what they can do well, or they can simply spend time doing more of what they already know they can do. An inexperienced programmer may technically be able to make an RPG from scratch, but it will be a very bad one by historical standards. Likewise, a very experienced programmer can still make a simple breakout-style or classic tank game of the sort that an inexperienced programmer might be wise to take on, but it could likely be done by the expert in much less time, with (virtually) no crash issues, and/or at a much higher level of overall polish. Naturally, if the goal is to both learn and produce good output, a compromise between these two ends of the continuum works best.

The last similarity to painting or playing an instrument is that, especially at a hobby level for personal enrichment and enjoyment, game programming is something that anyone with

enough patience can learn to do. Not everyone that paints is going to make works of art that sell for huge sums of money – very, very few will – and likewise the majority of people that play an instrument will likely never do so professionally. We're quite used to painting and instrument playing being a healthy part of life without needing to see careers defined by them, for example hanging up paintings by our relatives around the home, or playing guitar in the company of friends. I'm eager to see the generation after ours find it increasingly normal to compete in the freeware videogame made by a close relative, or to hang out on weekends playing a videogame made by one or more of those friends playing.

Of course if someone does have a professional interest in game making, whether at a company or going it alone, they are much more likely to build momentum and relevant skills by creating videogames as a hobby than by

simply talking and thinking about it.

This goes back to the opening question. Professional videogame programmers – including lifelong programmers with world-renowned accomplishments in the videogame industry – are still in the process of learning game programming, in the sense that they are continually learning new practices to make the most of new technologies. Someone is never done learning game programming until they are completely done with programming videogames, since the very act of programming videogames always involves some experimentation, digging through APIs, and solving problems that are new to us. Even when the problems that are new to us have been solved before by someone else, for all but the very hardest or most general of problems we can often solve them on our own with less time and effort than it would take to dig up and adapt someone else's solution.

But, roughly speaking, I'd offer the following estimates (remember that this is time to learn how to do something, not the time it takes to do it once learned) -

Recompiling and running existing sample code: 1 day.

Being able to tweak someone else's code: a few days, to a few weeks.

Being able to significantly add to or change example code: a few months to a few years, depending on the complexity of code being altered, or the size of the change being made.

Here there is already a massive dovetail of differences – adding a weapon to someone else's 2D sh'mup example code is a very different undertaking than adding network play to an open-sourced single-player 3D game. To continue, though -

Being able to write a crummy game from scratch: a few months.

Being able to make a decent game: 1-3 years of practice.

Ability to create something a stranger might like: 2-5 years.

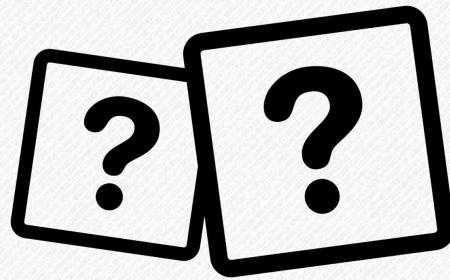
Ability to create something strangers might buy: 5-15 years, though obviously varies based on the price, the strangers, the project, the presentation, and a million other factors outside the programming itself.

Note too that due to the team nature of game development beyond the smallest scale, it's somewhat possible to "import experience" by partnering with someone that has been doing it longer. Working with someone that has more practice with videogame creation can improve the overall quality of work coming from everyone on the team, due to the added ability for the team as a whole to detect and avoid common issues, to employ tried-and-proven strategies that have worked out on previous projects, and to build while accounting for

how the big picture will need to come together by some fixed date in the future. When working alone, by comparison, those skills typically need to be learned cumulatively by working first on simple enough games that the light at the end of the tunnel is visible from the moment the project starts, then gradually incrementing project complexity in a way that the lessons learned from past projects can be built upon for the next.

HOBBY GAME DEVELOPMENT: 20 QUESTIONS

I wrote an article like this one for Game Developer magazine's Career Guide. If I had only one entry to put in front of people, this had to be it! Here is an extended version, republished here with permission.



For my first decade making videogames, I was a hobbyist. I created dozens of freeware games without any intention of making my living from it. Since then I've found joy in helping others get their start in hobby videogame development. A handful have gone on to be professionals in the industry, but many are content to just be making videogames as a pastime.

Alas, certain questions seem to frequently pop up that are preventing a whole lot of people

from getting started, trapping them from progressing, making it hard to finish projects, or leaving them unable to get the most out of their games once finished. Because so many of those same questions seem to appear for so many different people, I've collected here the responses that seemed so far to have best helped the people I've worked with.

Let me be clear that these aren't just guesses or theories. In addition to my own volume of work I've also seen hundreds of

developers getting into small-scale noncommercial videogame making with various attitudes about it, some of whom fall off the rails at various stages, while others roll on project after project.

Questions About Starting

Out of the many people that are capable of making videogames, that want to make videogames, why are so many of them not making videogames? These are the questions I have heard people explain as blocking them from getting started.

QUESTION 1: HOW CAN I RAISE THE MONEY NEEDED?

Answer: AAA videogame development can cost tens of millions of dollars. Hobby videogame development can be done entirely for free. Think of commercial videogame development as architecture and construction; think of hobby videogame development as making pillow forts out of anything in reach. If you are spending

money on making your pillow fort, you are doing it wrong.

QUESTION 2: WHAT'S THE BEST PLATFORM, DISTRIBUTION CHANNEL, OR STRATEGY TO MAKE MONEY?

Answer: Has this question ever kept anyone from picking up a basketball, learning guitar, or taking dance classes? When, in rare circumstances, people do wind up doing those things professionally, they aren't doing it for the first time when someone offers to pay them to do it – they did it for years out of a love for doing it. Art and shop classes don't begin with an introduction to marketing, not only because that's a fundamentally different skill, but because overemphasizing money early on can spoil the craft.

QUESTION 3: WHAT IF I'M NOT A GOOD ARTIST, OR PROGRAMMER, OR DESIGNER?

Answer: In that case – and this is usually the case – you will have more to gain than someone who is already good at those skills. Practical application is a great way to learn. The goal initially is just to be functionally capable of doing those tasks, in order to create

something from nothing. In the same way that anyone can draw or sing (even if not incredibly well), anyone can build a level, anyone can make a 2D sprite, and anyone can learn enough programming to at least make something simple. After that, practice will lead to making things that are increasingly more involved.

QUESTION 4: IF I CAN'T MAKE THE NEXT HALO OR FINAL FANTASY, WHY BOTHER?

Answer: You have numerous advantages as an independent developer that the huge videogame studios don't.

You can see your own ideas come to fruition. You can follow your gut. You can play around with designs that would be far too crazy to bet on if the financial yield had to recoup 3 years of full-time employee salaries. You don't have to prove anything to anyone except your players (and/or maybe yourself) – there's no asking permission or trying to win anyone over.

Enjoy your freedom, and find ways to make the most of it!

QUESTION 5: WHAT IF MY HEART IS SET ON WORKING WITH HUGE GAMES, AND MAKING A CAREER OUT OF IT?

Answer: Another way to get started, that can help meet this frequent request, is by modding. This option may work out better for people interested in specializing in modeling, audio, level design, and genre-specific programming. Before I was creating games from scratch, my first development experiences were making tweaks, weapons, and levels for Command & Conquer, Doom, and Descent.

Not every commercial game is feasible for modding. However many of the most popular PC games from the past 15-20 years have great tools and active communities. This is a fairly different subject than hobby game making though: search the web for modding. You'll find plenty of great material and tools online.

Indecision Questions

Buridan's Donkey is a parable from philosophy in which a donkey, standing before two equally appetizing bales of hay, was unable to decide on a reason to eat from one pile rather than the other. Paralyzed by indecision, the donkey starved to death, next to more than twice as much food as it needed.

In other words: any answer to these next questions is better than no answer. Pick one and run with it.

QUESTION 6: WHICH PROGRAMMING LANGUAGE SHOULD I USE?

Answer: To develop very smooth, high-performance downloadable computer games, C++ (using SFML, Allegro, SDL, or DirectX for the graphics, sound, and input) is still performance king. To make smaller games that can be played on the web without players needing to mess with installation, both HTML5 and ActionScript 3 – which compiles to Flash programs – can be used for free without

Adobe's Flash CS animation tool – can serve that need. Modern console and PC games are almost all programmed in C++, whereas ActionScript 3 and HTML5 projects dominate casual web game sites. Unity has also been gaining a lot of ground very rapidly in enabling solo developers or small teams to make impressive, cross-platform 3D and 2D games, and is well worth learning.

But here's why I brought up Buridan's Donkey at the start of this section: the only wrong answer is no answer! Don't be the donkey that starves to death in front of twice as much food as it needs, don't be paralyzed into indecision by there being more great and valid ways than ever before to make videogames. Get started on any of them, be ready to learn a lot as you go, and even if you find that you change your mind later for the next game, a lot of that same learning and process will transfer smoothly into the next platform you work with. Pick one

that you can find at least a few finished games were made with, and get started!

QUESTION 7: WHICH TOOLS SHOULD I USE?

Answer: I'm partial to free tools that get the job done: I suggest GIMP for 2D images, Blender for 3D models, and Audacity for editing recorded sound effects. In Windows, I program C++ projects using Bloodshed Dev-C++ 5 (though for more recent developers Code::Blocks has largely taken its place), and ActionScript 3 projects using FlashDevelop. On Mac I tend to get by with TextWrangler and recompilation scripts.

QUESTION 8: WHICH PART OF MY DREAM GAME SHOULD I WORK ON FIRST?

Answer: If it's really your dream game, and you want it made right, you won't make it the first project that you work on. I recommend getting a healthy chunk of beginner errors out of your system by making a few very modest and simple projects first.

There are two strategies for thinking about scope that I have found effective in guiding initial planning:

1. The Console Decades Ladder

Make something of '70s complexity first – on the order of Pong, Breakout, or (if you're feeling fancy) Missile Command. Once you have one or more games of '70s complexity behind you, move on to one or more games of '80s complexity, before going on to '90s complexity, and so on, sticking at whenever phase you prefer.

2. The Demo is the Game

If you made a shareware/demo/lite version to show off what the game is about, in an effort to rouse excitement about the full product, what would need to be in that version? Plan on completing the game at that scope as the final draft. If it comes out well, you can build upon it for a longer or more involved follow-up.

QUESTION 9: ARE SOME GAME TYPES SAFER THAN OTHERS EARLY ON?

Answer: In my experience overseeing and assisting with student projects, RPGs and side-view character-based genres are the most common failed or cancelled first-timer projects. This is due largely to their hefty art requirements.

Design the game with your art quality and quantity constraints in mind. Some 2D genres – such as overhead racing, side-view flying, abstract puzzle, and overhead space shooters – require virtually no animation, such that merely rotating and sliding static images via code will suffice. The other major benefit to these game types is that they often do not require hand-designed levels, instead spawning opponents semi-randomly, which saves the development effort required to create a level editor, level format, and levels.

Note too that online multiplayer is often much more complicated to

build and test than a local-only game. Sticking to single player projects, or videogames where players can share the same screen and keyboard, can greatly simplify technical matters. To provide a sense of scale for what can be involved, for several of the student and hobby projects that I've seen support online multiplayer, they first spent a full semester building the game for single player with AI opponents, or local multiplayer, then still needed an entire second semester – completely doubling the project's development time – to focus on getting online multiplayer ready for release.

QUESTION 10: WHAT SHOULD I PUT IN MY DESIGN DOC?

Answer: It's a trap! There's no right answer to this question, because I think most hobby videogames should not have a design doc.

Starting by writing a 20-page design document is silly, and usually a misuse time for a beginning developer. This is like someone that has never cooked

anything before trying to invent a recipe in Microsoft Word, or someone that has never worked with wood before trying to invent a woodworking project by sketching. As soon as the actual project starts, that plan has to be either thrown out (a waste of past time) or constantly updated (a waste of future time).

Use an image editor (GIMP, Photoshop...) to make a rough mock-up screenshot showing how the game might look when running. This will get you thinking about pixels and user-experience. Then write just enough code to bring the screenshot to life. If writing something down helps, I'd recommend keeping the treatment to under one page, and focusing on notes about gameplay rather than the game's fiction.

Project Finishing Questions

Unfinished videogame projects are not victimless. They waste the talented work of anyone that helped make content for the

project, they cheat players out of an opportunity to explore someone else's imagination, and they cause a severe morale hit to the developer(s).

Fortunately, just a few causes tend to be responsible for the overwhelming majority of sunken hobby projects. That means that each can be addressed on its own here:

QUESTION 11: WHAT CAN BE DONE TO KEEP STRESS AND SOCIAL POLITICS FROM PULLING THE TEAM APART?

Answer: Keep the team as small as possible, to minimize the probability of severe personality mismatch.

Minimize the project's duration to several months, or ideally even less for the first project or two. Any rough tensions, frustrations, and misunderstandings that might be present between people on a team are likely to grow if a project drags on.

Going into a project as a lead, make sure that you're ready and able to functionally (even if not as

well) complete any of the kinds of work that you're bringing aboard team members to help out with. Your ability to fill in for missing areas to keep the overall project on track gives other developers on the team the confidence that no matter who else leaves the team, the show will go on and their work will appear in a finished game. This confidence can help keep others productive.

QUESTION 12: ARE 5 PROGRAMMERS, 5 GAME DESIGNERS, AND 5 ARTISTS ENOUGH FOR 1 GAME?

Answer: If nearly everyone involved is a beginner, then that's far too many people for 1 game.

Bigger teams lead to more ambitious plans, longer project schedules, more communication challenges, more development bottlenecks, and more conflicts of both style and opinion.

I've seen student hobby projects with 25 members give up, as the project leader claimed that the team wasn't big enough to finish, while a half dozen teams of 2-4

members all completed their games.

Encourage the group of 15 people to split into 5 teams of 3, and overall you're more likely to see a yield of 3-5 games, instead of 0. Don't put all your eggs in one basket, avoid diffusing the clear responsibility of one person ambiguously across multiple people, and aim to create an atmosphere in which teams can learn from one another's independent investigation or experimentation.

QUESTION 13: HOW CAN A PROJECT BE PROTECTED FROM FEATURE CREEP?

Answer: When the game is in finaling mode, beginning perhaps halfway or three quarters through its scheduled development time, generate a list of the bare minimum that needs to be done before the game can be considered finished.

Now here's the important part: only allow that list to shrink!

There are two ways to shorten the list: complete the item it refers to,

or cut the item. Whenever you can make a cut without decreasing the quality of the game – even if only because it allocates more time to doing other tasks that must be done – make that cut.

If this sounds ruthless or contrary to the idea of hobby game making, note that until it's finished, it isn't a videogame. A garage full of carved up scrap wood is not the sign of a woodworker – a homemade spice rack and a new chair are. Just as the outside world does not want to drive a 90% working car or live in a 90% finished house, no one wants to play a 90% completed game. The other catch is that until it's done, it's impossible to even say what "90% done" really means; sometimes we have to start over to fit all the pieces together, which puts a perceived 90% much closer to 45%. It's either done, and a videogame, or it isn't, and it isn't. As the old Apple Computer line goes, "Real artists ship."

QUESTION 14: WHEN SHOULD WE STOP POLISHING THE CURRENT IDEA?

Answer: Iteration – the cycle of tweaking, trying out, and repeating – is an important part of polishing every game. However, tuning offers diminishing returns, adding less overall value with each pass. It doesn't take long before the game is as good as it's ever going to be, the tweaks are too minor to affect user experience, and it's time to wrap things up in order to advance to the next idea.

The old 80/20 rule of thumb suggests that 80% of the player's attention and enjoyment comes from 20% of your work. If you can't find something to tune that's likely part of that 20% the player will notice or care about, test the game a few more times from start to finish without introducing any new changes, then call the game done and move on to another.

QUESTION 15: WHY DOES "FINISHED" WORK KEEP GETTING THROWN OUT?

Answer: This happens when a project's decisions are being made in the wrong order, causing

everything to change whenever anything changes.

If the main character's jump height is changed late in development, that will potentially break every jumping area in the entire game. Until enemy health, weapon range, and movement are finalized, levels should be throwaway test cases; after those decisions are finalized, and levels are made based on them, those values should not be revisited. Aim to make decisions in an order that minimizes thrashing.

Finished Project Questions

What is one to do with the game after it's done? Read on:

QUESTION 16: WHY AREN'T MORE PEOPLE PLAYING OUR GREAT GAME?

Answer: First and foremost: do some marketing. Use Screenflow on Mac or Fraps in Windows and cut together a simple gameplay trailer with some music over it. Tip for that: record with music disabled in the game but sound effects on, then for the trailer add different music and decrease sound volume, so the video won't

jump in music or be without sounds. Make sure people know about it. If it's something you're pretty proud of, enter it in a competition. Blog about it. At the very least tweet and post to Facebook about it, tell your friends, share it with online communities. Don't just finish it then shove it in a directory to gather virtual dust. Games are meant to be played. No matter how great it is, no one will download it or even visit the in-browser URL if they don't know what it is, and how to get to it.

Assuming that you have achieved some visibility on the project, the other possibility is that people are having a hard time getting it or running it. In many cases, this happens because the project isn't well explained on the web, was packaged improperly, or wasn't tested on alternative browsers, operating systems, or machines.

If it's downloadable, either set up a single-file installer, or create a nicely organized zip file with

ReadMe.txt/PDF instructions inside it. Try running the game from a few different computers, to verify that the package you're giving out to the world isn't assuming a particular library or framework is already installed. Present the game's online link alongside a description plus a few screenshots.

If it's a Flash-based game programmed in ActionScript 3, consider posting to sites like Kongregate or Newgrounds. This often involves little more than creating an icon and writing a short description, and can get a few hundred plays minimum, thousands in most cases, and sometimes tremendously more.

After spending months on a game project, spending even a few more evenings making it presentable can have a dramatic impact on the number of people trying the game. This is time well spent.

QUESTION 17: HOW CAN I KEEP MY MIND ENGAGED BETWEEN PROJECTS?

Answer: Game making is a form of speaking, and the time between projects is an ideal time to catch up on listening. Our minds can fortify mid-development, filtering attention based on "does this apply to my game?" Tear that filter down, let things in, and expose yourself to some new ideas.

I strongly suggest carrying a pen and a small notepad, to jot down thoughts – not to save them because they're valuable, but so that you can be free to move past them. You'll encounter and come up with a ton of mediocre ideas throughout the day, and they'll get stuck in mind until you let them out. Simply putting the gist down on paper can free your attention from juggling it, making it easier to move on to finding and considering the next ideas.

QUESTION 18: IS THERE ANY WAY OUR GAME COULD HAVE BEEN BETTER, GIVEN THE SAME TIME AND EFFORT?

Answer: Don't spread the development efforts as thin. Focus the love. The same amount of

attention that could be poured into a poorly-made medium-sized game, could instead be applied to a well-polished small game.

Remember too that any partly unfinished game could have been a completely finished slightly more modest project.

Control scope every step of the way: plan with it in mind, develop with it in mind, end with it in mind. If great ideas come up during development that won't fit in the schedule, record them for possible implementation in either an update, sequel, or later project.

QUESTION 19: HOW IMPORTANT IS IT TO GET EVERYTHING THAT I MADE INTO THE FINAL GAME?

Answer: Films and novels are edited aggressively, and much to the benefit of the end result. Most videogames probably should be edited aggressively, too, and this can include chopping out parts that are finished. In game design, if the inclusion of something is not improving the overall experience, then it's getting in the way of the elements that do.

Now, if it's a student project, and it's just friends working together to get experience on a hobby project, then it's definitely worth considering ways to include at least some work from all contributors if possible, especially those that genuinely committed themselves to doing their best possible work for the game. If something is bad enough though, at least try to figure out some way to get that developer to do another iteration on it, or arrange the game in such a way that it won't block / gate access to content by other developers on the team. If it's really, really bad and additional iteration can't seem to fix it, see if they're open to the possibility of making something else from scratch to replace it. If the kind of work they were doing just wasn't coming out at usable quality (ex. their character art just isn't coming out right), see if they're open to contributing in some other way (assistance with sound? leading some external

playtesting efforts? something!). At the end of the day though, recognize that an awful level or puzzle or encounter early on may prevent players from experiencing the other content later in the game that came out well, and fairness to an individual developer many need to be balanced with fairness to the team as a whole.

If it's all your own work in question though, because you developed solo, don't be bashful about ruthlessly cutting the worst parts away before release. Is the only part of the game that really came together well one of the minigames? Consider carving away the rest and just releasing that minigame. Seriously. Or at least separately, in addition to still also releasing the rest of the game (heck, for an interesting little experiment, pay close attention to how responses vary to the two).

The best 30% of anyone's ideas or work are better than the other 70%. This seems obvious, yet so few people act on it. Think about it

like a simple English paper for school, something we all learned in middle school: there's more to do in going from a first draft to a second draft than fixing spelling and grammatical errors – for videogame projects most people skip the revising and focusing altogether as if their first output was a second draft and they're ready to fix up little bugs then call it a final draft. Ever play a game that you would have recommended to friends, if only "they had cut that one part out?" Find that part in your game, and be brave enough to cut it!

Great developers make self-editing part of their process. Strive to create more content than the game is intended to have or than it needs to have, and going back to trim out their worst material to leave only the best. Need 12 levels? Make at least 15 to choose them from, or if you want a better set of 12 levels, make 24. (Then resist the temptation to pollute your end result by including at the

last minute the very garbage that you consciously decided to throw out.)

QUESTION 20: IS IT POSSIBLE TO KNOW HOW OFTEN MY GAME IS PLAYED, AND TO PARTLY RETAIN THAT AUDIENCE?

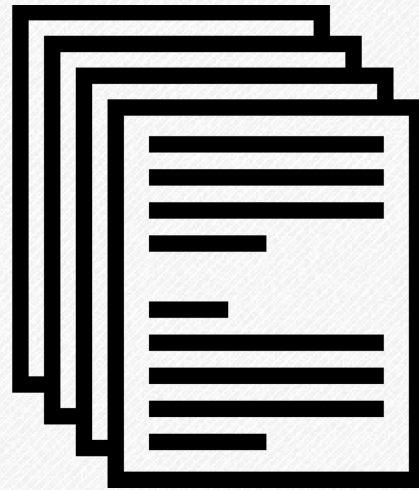
Answer: Keep metrics. Use StatCounter or Google Analytics to keep track of how many people visit your game's website. You might be surprised which of your projects gathers the most attention. Whether you're sharing your excitement with family members, trying to recruit new team members on a later project, or updating your resume in an effort to go professional, it would be nice to later know whether your game had 3 or 3,000,000 plays.

The second piece of that question is to link back to your site, either via a menu button in the game or through a shortcut in the game's start menu folder or at least include the URL someplace in-game. This enables satisfied players to find out more about your current and future work. Even if there isn't much on your site

now, there may be in 5 years, when you're still pulling in traffic from old games you made back in 2014.

BEGINNERS SHOULDN'T START WITH A DESIGN DOCUMENT

I've seen many people try to get started in videogame making by writing a huge document. As soon as work begins on the game, plans change entirely! Documents can be useful... but far more so after getting experience.



If you wanted to become a chef, would you begin by writing down recipes on paper?

I would start by following recipes. Then I would modify the ingredients or cook time in my favorite recipes, experimenting in the kitchen. Only after a great deal of experimenting would I consider inventing a new recipe starting on paper, since it would then be informed by grounded consideration of how various ingredients combine and react to different preparation processes.

If you wanted to become a singer and songwriter, would you begin by writing notes and words on paper?

I would learn some chords, and take vocal lessons. I would start performing some popular songs that my friends and I would immediately recognize. I might write my own spoof lyrics to the tunes I know, add my own flair to the notes, or read up on music theory. The first real inventing would likely to involve plucking the strings or playing the keys, and

experimenting with how words come together in my voice. Only after a great deal of experience inventing directly with instruments and voice would I have a sufficiently grounded understanding of how music works to put notes on paper as a first step.

So why on Earth, when most people want to become a videogame designer, do they begin by writing design docs?

Doesn't it make more sense to...

1. Play lots of videogames. For years. Learn what's out there.
2. Maybe mod some existing games, commercial or open source or just tweaking example code, to try putting in new weapons, levels, options and characters into videogames that you and your friends can play.
3. Program clones or pieces of simpler videogames that you like (classics, or casual webgames), nearly exactly, to learn about how

the design ideas translate into program code to form a finished project.

4. Start programming and designing variations on games that you like – by structure, extension, or adding personal touches.
5. After gaining comfort with this, experiment with inventing your own videogames, doing so with programming and image creation, and only jotting notes as needed to keep your thoughts organized.
6. Only after a great deal of experience creating games with code and asset creation, will you be remotely capable of inventing a new videogame starting on paper, based on your experiences with how various gameplay elements and content work together, with consideration for their impact on schedule and talent needs.

Is anything stopping you from writing down a recipe when you've never cooked before? Or putting notes on a staff with words

underneath, if you've never touched an instrument or sang a song? No. Of course you can do that. All it takes is a pencil. But it will be absolutely terrible. The same goes for videogame design. Except that unlike those other examples, a bad design doc can burn months or years of a lone developer's time, or from the lives of a team of people, to yield something that either won't get finished or that members of the team won't be proud to show.

I'm not trying to discourage anyone from starting. I've devoted my life to making videogames, and helping others do the same. I'm only trying to warn gravely against a common first step that I fear may do damage that takes years to reverse.

Keep your cooking in the kitchen, following recipes at first. Let your instrument and voice guide the music you compose, but only after learning classics or hits.

CLONE VIDEOGAMES TO LEARN REAL-TIME VIDEOGAME DESIGN

This is the most universal advice out there for people new to game development. However many people feel guilty about first trying it – unoriginal, immoral, etc. But it's just for practice! There are reasons for it. It works!



Is there a videogame you enjoy that uses simple graphics, relies more on action or puzzles than high fidelity art/audio atmosphere, and can get by with straightforward level structures and (if applicable) AI?

Clone it. Or at least most of its core functionality. Not as a business move, and not to try to get famous off it, just as an exercise in studying the details of its gameplay.

By cloning a game that you already like, you will...

1. ...be pulled into studying the concrete ins and outs of an existing game in great detail.
2. ...have a clear, complete, and coherent target to compare your work against.
3. ...avoid being stalled by uncertainty about what else is left to be done.
4. ...skip the common beginner risk of stalling until quitting from

lack of knowing when/how to end the design phase.

5. ...focus first on how to make things work, without being distracted by deciding what to make work.

6. ...know for certain that the skills you're practicing are immediately applicable to the sort of videogames you enjoy.

I'm not suggesting this as a way into becoming a programmer – I'm suggesting this as a way into becoming a better designer.

People can talk for a long time about how much better something in one game feels vs another (combat, navigation, camera, level layout, pacing, difficulty, power-up design...), but until someone can actually make something that feels the way in question, they're likely overlooking the most critical aspects in favor of the few qualities that are easiest to verbalize.

As the clone nears completion, there's room to experiment with

making it your own. The goal doesn't need to be per-pixel recreation of the original, but instead a careful understanding of it, in the interest of advancing our understanding as designers.

Sometimes that can best occur by contrasting your own implemented ideas to the workings of the unchanged version. (When our Great Ideas for what we thought ought to have been done differently turn out much worse in practice than the original approaches: bonus learning!)

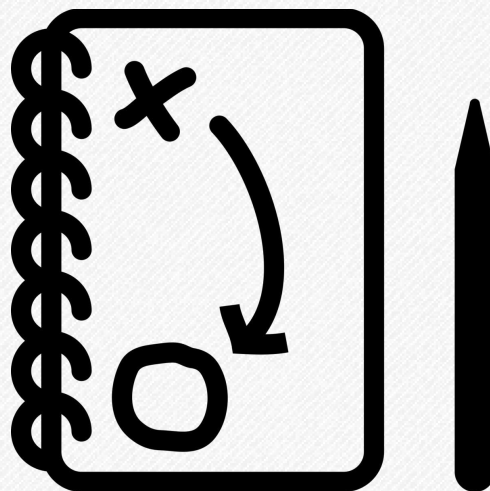
With practice and variation, this translates to being able to express your ideas to others by making the ideas come to life. This leads to becoming more effective as a standalone developer, should you choose that route, due to knowledge of how to make each part for yourself when needed. It can also make someone a more effective and articulate team leader, on account of the gained sense for how all the pieces fit together.

It's a worthwhile investment of time to do this just to share the results with friends, or to enjoy them for yourself, but most of all for the learning. In the future, when taking on more ambitious projects without a clear, predefined target, you'll be able to accelerate exploration by Frankensteining together source code, strategies, and learning from your practiced clone projects.

(Disclaimer: I do not propose nor condone cloning for the development of commercial videogames. By the time people are experienced enough to make videogames to earn money, they ought to be capable of coming up with and developing their own ideas.)

GENERAL CONCEPTS FOR BEGINNING DEVELOPERS

Some tips, while useful, are too small or weird to come up as part of a larger, structured class or book. Accepting that, I've collected here some helpful tidbits that may not fit together to form a larger common idea.



Often, in covering the specifics of examples, the more general issues and strategies of videogame development get overlooked. In this chapter, we'll focus on those general issues and strategies.

A scattershot approach is taken to cover a lot of information here – by all means, skim and skip for what's relevant to the sort of challenges that you're running into or thinking about. Very few of these sections are order dependent or cumulative. Broadly, I have grouped the points under

Design, Programming, and (a fairly cursory mention of) Math for videogame creation.

Videogame Design

THIS IS AN ART – IT'S INEXACT

Videogame design is more like writing a song than solving a math problem. There's not really a right and wrong to it. Personal preference also goes a long way here – just as the best techno song in the world still won't please people that don't like techno, a

great strategy game may seem boring and obtuse to people that don't like strategy games. That sort of differences in preferences is not only OK, it's unavoidable.

That said, everyday experience makes it pretty clear that everything is not equally good. Out of the seemingly infinite combinations of noises that could be created as "music," a comparatively narrow sliver of those possibilities can be reasonably defended or commonly recognized as a competent work of music – we expect patterns, we expect beat, we expect harmony, we expect emotion, and so on. In the same way, there are expectations that videogame players, inexperienced and experts alike, bring to their playing, and good design often involves being aware of those expectations, and only violating them if there is a rare and compelling reason to do so.

DESIGN THE VIDEOGAME

The end product is an experience for the player.

Not a design document to publish.

Not source code to hang on the wall.

Not a list of rules to be discussed.

Not a spreadsheet with elegant balance calculations (Side note: I'm skeptical that these have much value in any case! Animation times, psychological effects, holistic strategies, etc. make the raw numbers a poor reflection of what most people will actually do.)

If what you're doing isn't contributing to improving the user experience of the final result, second guess the value of it.

Don't confuse the dance steps for the dance.

80-20 RULE – "BANG FOR YOUR BUCK"

To reiterate the classic 80-20 rule of thumb about design: 80% of the user's enjoyment comes from 20% of the work that you produce. How the player controls,

especially for a real-time game, is often within that 20% – if it's done poorly, nothing else about the game is going to save it (see: Lair for PS3).

Even though it's impossible to tell exactly what falls within that figurative 20% the player is most concerned with, strive to get that fraction the best it can be.

Whatever falls in that figurative 80% should generally be just good enough to not distract from the 20%, otherwise it's sucking time and energy away from focusing on that core. Every paragraph in a novel doesn't need to be tied for being the greatest paragraph ever written for the book to come out well. At the same time, any book filled with typos and distracting grammatical errors won't be given a fair chance, no matter how clever it is.

“Bang for Your Buck” trade-offs are a useful way of thinking about videogame design. Even if you're not on a financial budget (as opposed to the time budget that

we all have to deal with), even if you're working alone and not on someone else's time, there are always opportunity costs when spending time on one part necessarily pulls that attention away from other parts.

Don't let pixels in far away background details, or trying to get a random scene midway through the game perfect, get in the way of getting right the player movement, camera feel, consistent visual language, and first impressions.

IF IT DOESN'T ADD TO IT, IT'S TAKING AWAY
Having more features is not better. It's different. In particular, to the extent that adding more no longer works well together, it becomes less focused, and feels like a less well designed experience.

I don't always edit my online writing as well as I'd perhaps like to – but at least in this case, headers make it easy to skim or skip any parts that someone finds uninteresting or irrelevant. In a videogame that luxury often isn't

there, since players are usually forced to take on a particular encounter, solve a particular puzzle, or otherwise grind through each planned experience before advancing.

One of the major differences between Google and Yahoo, since the beginning, has been that Google presents just a search page with a search bar, whereas Yahoo has always thrown 10 million irrelevant things (weather, news, entertainment, horoscopes, personal information, travel...) in front of the user. Yahoo is less of a disaster than it used to be – it used to look the way that (at the time of this writing) excite.com still does – but needless to say, the cleaner experience is widely preferred.

Show some restraint.

Also, don't be afraid to edit out of a project something that is already there, just as you would with a written paper, if it improves the overall effect. Cut things that are

confusing, that weren't taking shape well, that feel unfair even after several rework attempts, or that perhaps made sense in the original design but no longer jibe with what the game has evolved into during development. Shorter and more coherent is better than a mess that's a little longer.

The worst part of a game is frequently the memory that prevents someone from recommending to their friends what was an otherwise excellent experience.

CONSISTENT VISUAL LANGUAGE: ENVIRONMENTS

If the player is able to climb a tree, the player ought to be able to climb every tree. Moreover, the first time that non-obvious (it's not a common convention, such as dying when falling off the screen) control mechanism is relevant in gameplay, the user's experience should guide the player toward discovering this insight, either through:

- Appeals to real-world visual parallels, for example by arranging branches in a way that resembles a semi-regular ladder rungs.

- Affordance hints, such as alternating nobs on the tree, spaced within arm's length of one another.

- Curiosity lures, such as placing a desirable power-up at the top of the climbable portion, visible from the ground, which suggests to the player that there must be a way to reach it.

- Visual clues of being in the player's plane of action, such as by being at natural saturation in an environment where most trees are darkened out in the background.

- Demonstration through non-player/enemy utilization, e.g. having enemies climb up or down the tree to reach the player, showing that it can be climbed.

Players become confused, stuck, or frustrated when they are expected to read the designer's

mind, to somehow know that this tree, or perhaps this door, can be interacted with, unlike all that were discovered before or after it.

Consistency of visual language extends into all corners of the imagery in a game – something that harms the player should suggest it will cause harm (via spikes, boiling lava, angry expression), a platform the player can stand on should have visual similarities to others that the player can stand on, and so on.

CONSISTENT VISUAL LANGUAGE: ENTITIES

A similar consistency of expectation is formed by players in regard to enemies and items. If the player has encountered a number of witches before, one witch should not suddenly appear which takes or causes twice as much damage as those that came before. There are, alas, at least three exceptions that commonly come up where violating that sort of convention is acceptable:

1. There is an obvious visual difference in the new, more

powerful witch (tinted red, or substantially larger). This is not really an exception of the convention so much as a time-effective workaround.

2. If the first time the character appears it is presented as a boss encounter (given its own health at the top of the screen, supported by minions, required to be defeated for the player to complete an area), then on future encounters as a recurring enemy its strength as a non-boss may be toned down considerably.

3. Minor variations, such as sometimes requiring an extra hit or two to defeat, are accepted in less iconic, more literal representations of human (soldier, criminal...) or monster (esp. zombie) targets. Requiring an inconsistent number of hits to bring down adds unpredictability, luck, and realism to otherwise mechanically consistent encounters. Note that in this case, any given opponent should have a random opportunity to be the one slightly stronger;

planted deliberately, this defeats the purpose.

HALVING/DOUBLING TO TUNE NUMBERS

How fast should the player move in the air, in relation to how quickly enemy shots move? How rapidly does the next puzzle piece fall? There are at least dozens, and usually hundreds or even thousands of little numbers that can be tuned in a finished videogame. Where do those numbers come from?

Tuning generally comes down to whatever “feels right”. Isolate the most important number first – say, player jump height and move speed – then count on that as a pivot while tuning other values.

Values that “feel right” are often found by halving and doubling/splitting – programmers will recognize this idea as “binary search”, but the gist of it is that throughout the code, you’ll find many numbers for speeds, sizes, etc.

If something seems too fast, cut its speed in half. If something seemed too small, double its size. If one of those values was too far in the opposite direction, split down the middle on the next iteration, and so on, until it no longer seems to warrant finer changes.

By making dramatic changes in values, this also helps quickly ascertain whether the number in question is the right one to be futzing with. Changing the player's lateral air control speed in tiny increments may take many iterations to reveal that no value seems right at the current jump height – a discovery which is made much faster if the air control speed is tested in a broader search pattern.

For particularly messy tuning challenges, it can be helpful to set up a debug mode in the game, displaying the current value on screen, and supporting a few keyboard shortcuts that adjust the number either direction (higher,

lower, reset to default). In this way, a better value can be found through experimentation with less compilation delay.

TUNE IN ORDER

Once a good jump height and speed has been determined, levels can be designed based on that jump ability. At that point, either the jump values should remain untouched thereafter, or it should be considered that messing with them may invalidate previously completed levels.

MAKE SOMETHING YOU WANT TO PLAY

Unless you'd like to form focus groups and wait for their feedback between every major change in the game's design, and unless you're highly certain that you have a good way to both know that the project is meeting a certain player's wants and will reach them, make something that you enjoy. That way you have at least one happy player, there's no waiting for feedback before changing or trying on new ideas (part of what's great about working

alone or on small, hobbyist teams!). It's easier to work on something when the main drive is that you're eager for it to be done so that you can have it.

Working on something that you're genuinely excited to play also increases the incentive to finish it, and finish it well. If you were just a fan of the game you're making, you'd wish there was something you could do to get it done sooner and better; if you're a fan and the developer, that's on you. It's also a good way to play to your strengths, putting your best effort in, which ultimately results in better portfolio work (if relevant).

GIVE AREAS THEMES AND PURPOSE

A map designed as a series of obstacles with a sky or brick background only goes so far. It may get all the gameplay right, but if it's not giving the imagination something to chomp on, you may as well be making a film without audio. An important part is missing.

Is this level in a reactor? Is this area based on a kitchen? Or perhaps this is the place where the enemy has recreation between training? These can inspire the creation of an additional decoration or two, which can go a long way in setting the mood. They may also inspire substantial differences in layout and enemy/item placement, as opposed to thinking of it as "another level".

Just like there are visual themes, the gameplay should also be broken into differentiated phases, when possible. Perhaps this part focuses on jump platforming, perhaps that part focuses on lock/key or torch/switch puzzles, perhaps this other area is pure combat. It's entirely unnecessary (and often unwise) to cram every type of gameplay imaginable into every game, but what range is possible and worthwhile within the engine created should be explored in focused turns, rather than spread evenly into a murky gray from start to finish.

DON'T PUT EVERYTHING IN LEVEL ONE

I used to do this, especially in my early modding work. I wanted to show off every enemy, every weapon, every feature, everything cool about the game at the same time immediately to make what I thought was a first impression. Quite the opposite happens – it reeks of amateur, it's sloppy, it's unfocused, and there's too much going on for a newcomer to make sense of what's there.

Pick a couple of enemies, a couple of items, a couple of neat features, and rearrange them differently for the first couple of levels. Incrementally build upon those in the phases of the game that follow. Whatever you did for the prototype or the playable proof of concept – if it existed as such – generally needs to be dialed back and split apart into earlier levels to warm people up to that level of mid/late-game intensity.

A story shouldn't immediately introduce every character and plot point within the first few pages,

there should be some space given to letting the reader get to know the characters and circumstances before more are brought in. Your features are your characters; write a well-paced story.

YOUR GAME IS TOO HARD

I still have problems with this, and I've spent half of my life working on it. By the time the game is finished, you've played it more than nearly any other player (probably) ever will. You know each aspect for how it evolved over development, you've played it 20,000 times with a variety of different tuning and layouts trying to figure out what worked, you know how the AI and weapon collision systems and power-up timers operate, etc. If any level besides the last one or two still challenges you as a player by the time the game is done, it's probably ridiculously hard for anyone else. Dial it back, then put it in front of someone else to see whether they get annoyed by it.

SYSTEMS-CENTERED DESIGN

Systems centered design – the bottom-up thinking that leads to levels designed as grid tile arrangements, enemy/obstacle AI as simple patterns, and non-human/non-character for player control (tank, spaceship, yellow crescent, arrows) used to be common in the ‘70s-‘90s for the sake of working within technology’s computational limitations.

Even though a modern computer (or console, for that matter) is now powerful enough to deal with arbitrarily complex level structures, adaptive AI, and high polygon finely textured smoothly animated 3D models, making that stuff takes a ton of time and special training, and usually leads to what is only one particular type out of many possible aesthetics.

Making a single room for Doom 3 from scratch may take more hours than designing an entire level in Doom. If you’re not (and don’t have access to the time of) a

talented modeler, expert in lighting and audio, etc. your Doom 3 room would look rough no matter how much time is put into it; by comparison, pretty much anyone could throw together a pretty decent (even if not great) Doom level, since it’s a matter of laying down some overhead lines, placing enemies/items/decorations, and picking/aligning simple single-layer textures.

All that, and Doom 3 is a worse videogame than Doom.

There are a lot more ways to shoot yourself in the foot designing or creating content to fill an area in Final Fantasy XIII than in Final Fantasy, Final Fantasy III, or even



“Look ma, no arms.” (And the poor fellow on the right doesn’t even have legs.)

(relatively speaking) Final Fantasy VII.

All that hard work gets burnt on trying to force what's ultimately as artificial as a cartoon to look as photorealistic as live film. Would photorealism improve The Simpsons, South Park, or Futurama? Of course not. This leads to the next point, an important design choice:

LITERAL VS CONCEPTUAL

In *Understanding Comics*, Scott McCloud points out that the less literal an image looks, the more universal, idea-oriented, and conceptual it is.

Graphics used to be simple because that's all the hardware was capable of doing – a few colors at once, blocky pixels, etc. Even though more complicated things are now computationally possible, however, there are still perfectly legitimate reasons to go for a simple style – and it isn't just about saving time.

The purpose of every picture is not the same, and many don't aim to faithfully depict reality. That videogames have gradually been able to come closer to visually, verbally, and viscerally reproducing reality has made that a distinctive advantage in differentiating products within the marketplace. But then came along Wii Sports and its 5 piece, 4-color plastic characters that have one-color spheres for hands. In case you haven't been watching the score, 3.5 years after its late 2006 release it's still #5 out of worldwide weekly sales for all videogames.

Wii Sports isn't trying to accurately depict bowling, baseball, boxing, tennis, or golf, either visually or in mechanics. It's instead about the simplest ideas underlying bowling, baseball, boxing, tennis, and golf. It has more to do with how we think about those things than it does with how those things really are.

That's more than just fine. People love it.

Simple isn't worse. Complex isn't better. Simple is substantially faster to make more levels, characters, animations, and special effects for. Complex requires a large team with a great deal of very specific and specialized talent coordinating efforts. Simple is conceptual. Complex is literal.

Most interesting to me is that conceptual – and thus simple – affords such a vastly larger range of perspectives and ideas to explore and present, compared to the literal – and thus complicated – representation of how things really look, how things really function, and how things really tumble down steep hills. When it comes down to it, there is only one way that everything fundamentally and really is, but there are nearly infinite ways that things are not.

Weekly Software Charts

Top 10, ranked by number of units sold. Select region:

	Worldwide	Americas	Japan	EMEA
Worldwide Top 10 21st August 2010				
1	PS3	A.C.E.: Another Century's Epis...		216,683
2	X360	Madden NFL 11		141,323
3	PS3	Madden NFL 11		126,814
4	PC	StarCraft II: Wings of Liberty		123,375
5	Wii	Wii Sports		119,992
6	Wii	Wii Sports Resort		95,776
7	Wii	Super Mario Galaxy 2		84,166
8	Wii	Wii Party		68,556
9	X360	Kane & Lynch 2: Dog Days		59,415
10	DS	Pokémon Heart Gold / Soul Silv...		56,713

The 4 games above it were at this time less than 1 month old. The 3 games below it are also Wii games with simple graphics. The one immediately after it is its sequel.

If Braid tried to be photorealistic in 3D, it would have required a substantially larger team to produce, and come out (almost inescapably) far less interesting. Instead, almost all of the enemies are the same, and the level elements are repeated functional and decorative pieces (as with the design of a classical Genesis or SNES platformer). It wasn't laziness, and it wasn't lack of

ability/resources to realize a full 3D world; it was using systems-centered design to massively simplify/accelerate production while focusing in closely on concepts and ideas.

Have you played Canabalt? Huge breakaway success, using clunky pixel art, low-fidelity music, and a single procedurally generated level.

I'm almost at a loss for how else to put it, but it cannot be emphasized enough: resourceful, conceptual, unrealistic style and clever design, not unlike the sort that 15+ years ago were required for videogames to function at all, can today save a tremendous amount of time and energy... while often producing significantly more successful and meaningful work.

IS THE IDEA A VIDEOGAME IDEA?

Frequently, someone comes into videogame development with an idea of a story they would like to tell. Is the particular story not, perhaps, better suited as a cartoon, as a short film, as a play,

or even simply written in well-edited prose? Certainly, the audience coming into the story could be forced to unjam a virtual door with bobby pin before getting to the next scene, or required to fend off hordes of zombies (again), although videogame qualities are often adopted at the expense to the story concept as a prominent source of frustration or distraction, rather than as meaningful rhetorical elements.

There are videogames that do story well, in the way that videogames handle story. It's worth considering, however, that a person wishing to do two things – tell a particular story, and make a videogame – may in some cases be most pleased with the results doing those as two separate things, rather than forcing both together.

If the best way to bring your idea to life is as a videogame – or at least it might be, though you're not sure – by all means, go ahead. I merely wish to insert an extra

step of consideration. If the idea would come through better as an animated short, a written short story, or maybe even a song, it's 2014: you can learn to make pretty much whatever you set your mind to. If you have a hammer, every problem isn't necessarily a nail; if you're specifically interested in using a hammer for its own sake, no matter how many screws need screwing, that still doesn't make it the right tool for those tasks.

PUT THE GAME IN FRONT OF A FRIEND

Don't tell them anything in advance about how they're supposed to play it. Don't answer questions, unless they really need it to advance, and then make note of it. When other people get to it, you won't have the privilege of getting them unstuck when they don't realize that up arrow opens doors.

Whatever you had to tell the stuck friend, add some sort of indicator or message to the game to tell future players on your behalf. It doesn't interrupt or break the

game's reality to do this – at least, not nearly as badly as being unable to figure out how to open a door or perform some other trivial, required task.

Programming

ILLUSION OF SIMULTANEITY

To anyone that has done any real-time game programming, this probably seems like a no brainer. To someone that hasn't, getting used to this concept is a surprisingly common barrier when getting started.

Everything happens one thing at a time. The code in every function goes from top to bottom, one instruction at a time. Even though it looks like dozens or hundreds of things are happening and moving at once, they're each getting a turn. They appear to be moving at the same time since the updates are happening very quickly. The game moves and draws all objects by calling a main or draw function 20-60 times per second, top to bottom, instruction by instruction.

Often, the last step in a given lap through the game's movement, input, sound, and other code is to update the screen (technically also happening pixels at a time as the memory buffer gets copied to screen buffer). In some environments like Processing or ActionScript this step is handled automatically; in C++ or most other environments it must be done explicitly. This step is how things seem to all be drawn and moved at the same time: though everything is drawn and moved one at a time, the result of all those changes only gets updated for the display all at once after all increments for the current logic frame have taken place.

CODE ORDER MATTERS

This sounds like an insane thing to mention to programmers, but especially if someone is new in the transition from systems code into the object-centric real-time space, the seemingly self-contained nature of objects can be taken for

granted as pure conceptual independence.

Sometimes bugs happen due to the right code being in the wrong order. No matter how well we try to group and organize our code to be self-contained, it's always possible for otherwise working code to produce unintended behavior due to being out of order.

One very easy mistake of this sort is if the background images (sky or level tiles) get drawn after the player, items, or particles every frame, it will draw over them before each screen update, making them invisible. All the drawing code can be correct, with just one instruction in the wrong order. This isn't anything to stress out about, as a lot of compartmentalized code will work more or less the same regardless of its order, but it's something to be watchful of.

ASSET DATA FILES

With rare exceptions, we don't program the images or audio. We use programming to display

images at certain coordinates, and to play sounds at certain times, but the images and sounds are usually created in outside programs. Images are often made in Photoshop or Gimp, and audio is often created or edited in Sound Forge or Audacity.

The assets (images, audio, 3D models, etc.) are just plain image and audio files, in whichever formats both your programming library and asset tools support.

GOOD ENOUGH IS OFTEN BEST

Collision between moving objects is rarely done per pixel or per polygon. Instead, distance is checked between two objects, or simple arithmetic is used to determine whether rectangular bounding boxes overlap.

For all practical purposes, it's generally a perfectly fine approximation. It's a faster and easier way to implement collisions.

Even if we had detailed collision functionality done for us though – perhaps as part of a library, or

built into an existing engine we're working with – for most applications we still wouldn't want it. Simplified approximations like this one calculate faster, require minimal additional authoring (ex. polygonal collision detection requires an underlying collision shape to be defined for each graphic), and satisfy the conceptual goal in a way people are quite used to. If we went to the trouble of defining player character collision through detailed polygons defined for every animation frame, it may introduce weirdness like being able to run over the top of an item without collecting it – more development effort to produce worse results is a silly trade-off.

PROGRAM A LEVEL DESIGN TOOL

Level design should be done by dragging and dropping, filling, by lasso selecting, copy pasting, saving/loading, etc. Not by programming. Time that the programmer spends implementing level layouts is time that the

programmer can't spend fixing bugs and adding features – plus it's time that the designer has to wait between iterating on ideas. Even if you're the both the programmer and the designer (as is inevitably the case for a solo project, and common for a small project where it's mostly art/audio coming from others), it still massively accelerates production effort, ensures systematic consistency, etc.

The easiest way to make a level building “tool” is to hijack the game's existing code, burying the tool functionality within it. Upon pressing F1 or some other out of the way key, toggle to a mouse-control, toss a panel of buttons and objects on the screen, and switch keyboard to controlling some handy key shortcuts. This can be stripped from the finished game, hidden in the finished game as an unlockable reward, or simply released with a tiny bit of “the level editor is rough... good luck!” documentation/help.

CHEATS ACCELERATE DEBUGGING

Add in invincibility, support a way to start or teleport to alternate locations (even if only by manually swapping around start position in a level editor), a way to max out items, ammo, powers, etc. If every time things are tested you have to play well enough to not die, collect a certain combination of items, and get to the right spot to test how, say, enemies climbing ladders respond to being hit by rockets – it's going to take forever to confirm a suspected bug fix.

BUILD QUICKLY, EXPAND WHEN/AS NEEDED

Often in videogame development, we have to decide between a lot of ways that we could spend time adding or making cool stuff. It's hard to know sometimes until something is implemented whether or not it will be as fun on screen as the imagination suspects. For this reason it's often practical to find a hacky or brute force way to implement a feature that takes less time, if only to see how it works before deciding whether or not to keep it.

This avoids sinking time into code that there's a decent chance may get ripped out anyway.

If the hacked feature works well though, it's fast enough as-is, and the unpleasant details of its implementation can be swept into a function or class that you don't need to look at much for other development, leave it as is until (if ever) there's a sound reason for redoing it.

MINIMIZE TIME BETWEEN CODE COMPILATION

Get a single enemy working before expanding it into an array of them and/or supporting multiple enemy types. Build a very plain sort of particle effect first – white spinning squares that burst out might be a fine starting place. Then adjust or add functionality as needed to support different needs of particles (fading for smoke, glowing for flame, directionality for thrust...). Get one power-up working first, perhaps the simple 1-Up, then introduce others as a variation on that one.

There's necessarily time while programming, between setting out to make something work and being able to see it work, during which time the code is in an incomplete, dysfunctional state. That time can be minimized by this method of building simpler first, then expanding out from that base. Minimizing the time between compiles and visual results is helpful since it prioritizes bug fixes while there's the least variability. In turn, that reassurance from the machine that you're on the right track minimizes cognitive load, freeing up mental resources to more easily juggle bite sized problems instead of trying to do several things at once.

SPECIAL CASES SHOULD BE SPECIAL CASED

“Special case” is often treated as a bad word by engineers, but an actual “special case” deserves to be special cased instead of generalized. When a game has each boss behave distinctly, rather than as a recombination of shared components, it gives them more

unique character; when a level has code written that only applies to that level, so long as it doesn't break the player's trained expectations of what to do, it can go a long way in making the experience less formulaic.

STATE CHANGE FUNCTIONS

Some functions, like those that set font color, or position and rotate images, affect code that happens after them. Rather than specifying tons of parameters with each call, such as indicating that the text alignment should be centered, the code has a single line that centers any text calls that follow it, until code is added to change text to left/right alignment.

These sort of functions also appear in OpenGL, the graphics library commonly used for iPhone, downloadable, or 3D games. Unlike code which is called to perform some operation while accounting for certain parameters, the state change functions are called to toggle switches or set

preferences which will affect any graphics calls that come after.

Math

THE SCREEN IS A 2D PLANE

The main thing needed from geometry is thinking about the 2D game's visual space as a grid plane, with (0,0) in the top left, and (screen pixel width, screen pixel height) at the bottom right. Note that in most environments, down is positive Y, which is inverted from the standard used in classroom geometry.

TRIGONOMETRY AND VECTORS

For all things angles and movement, trigonometry and vectors come in handy. Though not covered in depth here, there are plenty of resources on the web to learn about these areas of math. When you spot them in sample code, don't gloss over how they're being used – they're often at the core of how the gameplay movements work.

MATRIX TRANSFORMATIONS (PUSH & POP)

Matrix calculations can be used to capture a series of graphical

calculations – scaling, rotating, positioning – into a single optimized transformation that can be applied to however many vertices or polygons need to be oriented as a set.

Because Matrix calculations are cumulative, this is a particular form of state change code. Each line accumulates to have a sum affect on all of the graphics code that follows. We can make a call to the translate function, giving it parameters for the camera's horizontal and vertical offsets, and it will affect all the graphics code which follows, which is quite handy. But rather than having to undo every slide, rotation, and scale operation that we call (this would be a nightmare for all those slipping, spinning, growing particle effects images!), we can instead “Push” a Matrix onto the stack, perform the translations and rotation that we want to apply to all following image calls, then “Pop” that Matrix off the top of the stack, automatically undoing that

transformation changes that we added after pushing it. Note that this only applies to graphical transformations – translation (sliding), rotation, and scaling – not to other state changes such as font color, image tint, etc.

LEARN VIDEOGAME DEVELOPMENT LIKE WOODWORKING

I took years of wood shop in high school. More important than a spice rack or picture frame, I took from it an approach to learning that I've since discovered works wonders for keeping game developers on a good track.



HOW WOODSHOP IS TAUGHT

1. The instructor first has everyone read a bit of text about shop safety (how to not lose an eye, how to not lose a finger...) and basic concepts (grain, sanding, estimating board foot requirements for rough sawn lumber).

2. A brief tour of the shop is given to introduce the various hand tools, power tools, and relevant shop features (light switches, ventilation fans, first aid kits).

3. Everyone does a series of simple projects, each of which involves 1 or 2 different kinds of tools. These are pretty much the same for everyone, possibly with 1 minor substitution. Common examples: a judge's gavel is made to learn the lathe and wood finish, and a simple picture frame is made to learn the router and joinery. For a younger or less comfortable audience, a birdhouse is a recognizable result that involves learning proper sawing and simple use of fasteners.

4. Once that series of beginning projects is out of the way – generally involving a semester and several finished pieces – a student is then able to bring in plans and suggest adaptations.

HOW WOODSHOP IS NOT TAUGHT

- Start drawing up blueprints for new ideas. To whatever extent drawing and the imagination are unrestrained by what wood and tools do, those activities are not woodworking – but to get a sense of that distinction first requires at least a little variety in hands-on work. Coming up with complicated, unrealistic plans is not woodworking.
- Build a complex, detailed piece of a larger project (such as an ornate leg for a chair). Being 100% done with 10% of something doesn't feel or look like being done, it feels and looks like being 10% done. The person on the project would not get a sense of completion, and it would not be in a state that it could be shown off or given away. Making 10% of

things then moving on is not woodworking.

- Subtle variation on the above: team up with 15 other people that have never manipulated wood or wood tools before in an attempt to coordinate building an advanced project (say, a small ship). 1/15 of that inexperienced team working on a 1/15 as ambitious project is significantly more likely to be finished, because it removes management complexity. Team management is not woodworking.
- Spend semesters making and discussing origami, under the guise that since both are construction from tree materials, there are similar principles at work. These are two completely different things. People skilled or interested in one may not be skilled or interested in the other. Papercraft is not woodworking.
- Teaching and testing the history of woodworking, investigating ancient tools and techniques or old uses of lumber that it no

longer fills today. The modern woodshop is very different from an ancient one, in terms of the tools and ease of material acquisition. History is not woodworking.

- Discussing how to market and sell a wood project. Marketing is not woodworking.
- Buying, using, and discussing the latest commercial products that are made of wood. Whether it's made by professionals and/or reflects the cutting edge of industry, it's not an appropriate starting point or reference point for beginners. Becoming a collector or connoisseur is not woodworking.

HOW THIS RELATES

Modern books on videogame design, many classes and workshops on game design, and other learning outlets focus on a number of the above bullet points.

Students often want to plan and innovate immediately, take on a big team project, or at least craft a complex piece of a never-to-be-

finished videogame (standalone inventory system, dialog trees, etc.). Teachers in many cases want to go into great length about board/card/dice games, or the history of games or play in general. Workshops and websites will typically go straight into discussion of ways to make money from a videogame.

Meanwhile student clubs tend to devolve into a gamer culture group that plays and discusses design of commercial projects that are far, far beyond the design, engineering, art, and management abilities of the club members.

The passion is an entirely good thing. Everything listed above is a legitimate field in its own right, bearing its own complexities and unique utility in the big picture, whether it's marketing, history, team management, etc. Origami is a fascinating and involved field with cultural roots and a tradition of great craft, but it is not woodworking. Likewise while board, dice, and card games are a

worthwhile and important subject with interesting histories and established conventions all their own, their relation to the design or development of real-time videogames is tenuous or genre-specific at best.

The woodworker knows the process end to end, from sawing, to shaping, to joining and finishing. Projects are identified that are appropriate for one person to make, working alone, in a modest period of time (generally no longer than half a year per project, until very advanced). The woodworker is not interested in the subject for the pure sake of knowing things about it, nor is the woodworker interested in the subject purely as a springboard into a career doing such work. Just as any woodworker can transfer knowledge of the jack plane to fix doors around the house that don't perfectly fit the frame, or otherwise apply tool comfort and familiarity to minor projects around the house, the

hobbyist/solo/indie game developer along the way picks up a myriad of skills and tools that can find application to meet the needs of future tasks (just as much for non-videogame related companies and non-videogame personal side projects).

A project or two that focuses purely on input, setting screen resolution and drawing rectangles, and doing collision detection (Pong) makes for a good intro project. Likewise, anything involving a small, simple tile-based world (either for bricks in breakout or a tiny dungeon world) can help with learning that method of level construction and collision detection. Building up from there, a similar mentality can be used to pick and adapt beginning projects to cover the skillsets and tools that one might be interested in applying later for a bigger project (loading and displaying images, playing music and sounds, programming menus, different types of AI, etc.), without needing

to jump directly into using the table saw on day one.

IT ONLY COUNTS WHEN IT'S DONE

Some people set mental goals in terms of identity, as in “I want to be a videogame developer,” or “I want to know how to make videogames.” The problem with these is that the first one is attained by simply starting, whereas the second one may never create a single videogame.

A more constructive way to think about it is in terms of completion of lasting artifacts – that is, “I want to have finished making a videogame” or alternatively, “I want to finish making my current project.” Molding and twisting reality to come into alignment with these thoughts will necessarily include accomplishing the two identity thoughts from the previous paragraph, the difference being that this one will also yield one or more games as evidence of that activity.

No one knows what you're capable of until after you've done it – not even you.

“We judge ourselves by what we feel capable of doing, while others judge us by what we have already done.”

-Henry Wadsworth Longfellow
US poet (1807 – 1882)

FAN HABITS AND FOCUS ARE NOT DEVELOPER HABITS AND FOCUS

There is confusion among people growing from being fans to being developers that they should still soak up all that they can through previews, reviews, industry gossip, awards events, new releases and new trends.



Consider whether a writer trying to stay on top of reading every top selling novel will be making much progress on a novel of their own.

True, reading books can tie up time, so let's also consider a song, which doesn't take nearly as long to listen to, and an image which can take even less time to look at. Should a musician listen to every chart-topping song by other performers? Should a 2D artist scramble every day to view the latest in popular illustrations and paintings?

The answer, of course, is no, in every one of those cases. They should not. A few reasons:

- Most music isn't going to be any given musician's style, and most artwork isn't going to be any given artist's style. In our case as developers, most videogames out there will have nothing to do with our genre, our audience, or our style. It's of course possible to glean inspiration from things that aren't in the same category of our own, but those cases are fewer and further in-between, and for

that matter could come from anywhere (including other mediums, or better, real life).

- Studying something, to extract something of use from its construction and presentation, requires a great deal more time and work than simply listening to it once or looking at it casually. It requires repeated encounters with it, deep diving into details, exploration to understand its context and synthesis, often going into information not apparent in the artifact itself, including the developer's other work and the business circumstances which either brought it into existence or pulled it from a fate of obscurity into the limelight. Dissecting even a 3 minute song or an image is no trivial task, and a videogame might take a month or more of focused effort to take apart. That level of sustained concentration on playing is easily incompatible with also giving production the consistent attention it needs.

- Just because something is popular, by any metric (financially, as a cult classic, or hitting the Reddit front page), definitely does not mean that it's any good. At the expense of sounding a little hipstery or jaded, a ton of commercially successful work in any medium is dumbed down, derivative, formulaic, and forgettable. In videogames, plenty of great work doesn't get much exposure because it doesn't come across in a screenshot and has zero marketing push, while other mediocre work is racing to the top partly due to strange market effects like established branding, nepotism between industry contacts, reviewer groupthink, and a whole mess of other factors quite other than the videogame itself. It's a mistake to assume that the best-known artifacts happen to also be the best ones for a developer to know.

Additionally, especially in the videogame case, if you're going to make a videogame based on

someone else's already popular work to ride part of its success (note: which is very different from cloning as a learning exercise, which is legit and very helpful), good luck. You're already a full development cycle behind your competition, and by the time you release they can be on a sequel or many updates later while countless others who had the same plan release at the same time as you.

Of course, I don't mean to suggest totally hiding from the world. Coming up for air is an important part of swimming longer. But if you're putting time into playing popular games or reading up on game news, make sure it's something you're doing because it's what you want to be doing, and not under some misguided impression that it's an important part of working on your game, or becoming a better videogame developer. In all likelihood, it isn't. If you really are deeply studying a specific game as part of

development, then truly investigate it: replay it, take notes on it, write about it, and once you've got what you need from it, move past it.

Instead of getting caught up in the news, make a point to periodically catch up on what went down. After the smoke clears. In a matter of days it's surprisingly easy to catch up on months of ignoring the news and new releases. By that time it's cheaper, too, on sale online or available discounted in the used and bargain bins, since full-time consumers have all moved on by then to the next newest thing.

Go get lost in making your own beautiful thing. Forget about the noise and hype, 24/7 consumer spectacle, the irrational rising and falling in the trends or reviews or stranger's drama.

The next game release you should be really pumped about is the one that you're currently working on.

2



I've taught game development to graduate students older than me, and I've also taught it to teenagers. Similar questions arise with both - it has nothing to do with age! It's just about what we've learned before. Here are questions that new students of any age may share.

QUESTIONS ABOUT EDUCATION

ADVICE TO A NEW STUDENT IN VIDEOGAME DESIGN

Students choosing to study videogame design in school occasionally write seeking advice. Here is my general response, with the hopes that some of it may apply to those making videogames outside of school, too.



Do more than is required. You'll get out as much as you put in. This means learning more than you're assigned, putting more into projects than is needed for an A, and making and releasing finished videogames aside from and outside of coursework.

Be okay with producing crummy results when you're new to something. Don't let that discourage you from sticking with it. There's a lot less competition in the world than people assume, due to attrition; most would-be

competitors will have given up long before getting over the barriers to entry, even if held up only by their own lack of confidence. Don't settle for staying crummy at it though. Practice, seek feedback, and strive to get your skills to a level that you'll have something to bring to the table when joining a team.

Try to not get wrapped up in talking about games instead of working on them. Make regular progress, even if incremental, and stick to deadlines. Sometimes this

kind of discipline means turning down social engagements on weekends. Don't totally deny yourself time around people either, or that can bring its own problems. Find your way out of circular or unproductive banter about the "best" design or related fashion - these are traps. Different audiences have different preferences about different matters, and it's almost always just apples versus oranges. Argue about game design instead by implementing your ideas and letting others play them, at least as playable prototypes if not within completely finished games. All the opposing words in the world won't matter much if what you've done just works.

Don't compare yourself to other people. That can only lead to mediocrity, plateauing, or hopelessness. Just do the best you can with things. If you need a reference point, try to end the day or week or month better than you

were when it started. Keep improving.

Network proactively. Go out of your way to be where people are that you want to meet, and go outside your comfort zone to start a conversation with them, even if it's a bit difficult or awkward at first.

Don't ignore feedback, but don't feel the need to act on all of it either. This will be hard but don't take feedback personally. It's about the project and the project is not you. Seek feedback, consider feedback, act on some feedback, don't be defensive of feedback or feel the need to argue about it. It's far better to hear complaints and frustrations while there's still time to fix them, before it's affecting strangers and you won't be there to justify questionable decisions.

Keep deadlines within reach – no longer than months away while new, and just a few weeks may even work better at first to get

some junk and early learning out of your system. Keep teams small enough (1-5) to have a clear idea of each person's responsibility and roles without stepping on each other's toes or added management complexity for planning and communications. A longer schedule isn't easier, and a larger team doesn't work faster. Both of those just blow up expectations and introduce whole new categories of challenges atop just getting the game working.

Don't get stuck in one platform, tool, or programming language because it's the one you know, especially not because it's the first or only one you know. It's worth learning new things and doesn't take nearly as long as people assume. Don't jump onto a new platform, tool, or programming language just because it's new or someone recommends it. It may be not ready yet or they may not know what they're talking about. Look for examples of what it has been used for and decide for

yourself if it's something worth getting into.

Focus on doing at least one tangible skill discipline very well, since that's what'll generally get you a job, but beyond that don't pigeonhole yourself from learning a bit of fluency in other aspects of development. Those other skills will help you communicate and work more successfully with people that have other roles and can empower you to pull together your own ideas as rough demonstrations without relying on the time or assistance of others, and can generally help you stand out in the workplace. Though, again, to get into said workplace you generally need to be especially solid in one core skill area since there aren't really official positions for jacks-of-all-trades.

Don't insist on doing something totally original immediately. Like an artist beginning by doing abstract painting or sculptures, that really won't go anywhere without some

credibility first built from practicing and demonstrating an understanding of fundamentals and technique. Trying to do something totally original and weird immediately just comes across in such cases as a lazy excuse for sloppy and undeveloped craftsmanship. Accept some major conventions and genres, at least at first and early on, so that you can focus on trying to do a good job of something that you know can be done well. Better a small and derivative but finished and polished game than an ambitious, highly original, unfinished and unpolished project.

Stay positive. You'll meet some cynical people along the way that may try to drag you down. No need to be upset with them over it. If they could, they'd often rather not be like that either.

Whenever possible, help, teach, or create opportunities for others that are less familiar with game development than you are. This

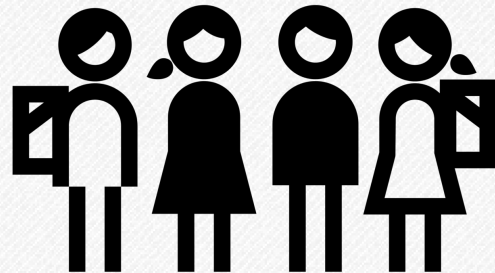
part of the population may be small if you're just starting out, but it will grow with experience. It'll often help you out, too, in the short-term because you'll have to review and learn how to articulate what you know, and in the long-term because as they gain different experiences and specializations, they'll later become your peers and potential collaborators.

Be nice to everybody. Stay out of fights, nobody wins.

Good luck!

QUESTIONS FROM AN ELEMENTARY SCHOOL CLASS

A teacher sent me questions from her students about videogame development. Kids ask the best questions, probably since no one is putting pressure on them by assuming that they ought to already know the answer.



HOW DO YOU USE YOUR IMAGINATION TO MAKE VIDEO GAMES?

To make videogames, I use my imagination to think of things that don't yet exist, or to think about how things that do exist could be different – combined, simplified, or built upon to be done better.

That's a "top-down" approach, meaning that I start with the idea and then figure out how to make the idea work.

Other times I instead use a "bottom-up" approach, meaning that I first create things that are

enjoyable to play with, and then I use my imagination to figure out how that enjoyable interaction or moment could be expanded into a presentable game.

HOW DO YOU MAKE PEOPLE TALK AND SOUND EFFECTS?

For voices, we typically record people speaking, then play back those recordings by loading and playing the sound file during the game. Many sound effects are also simply recordings that are played back when the right conditions are met. Some sounds

are made by starting with multiple recorded samples then modifying and mixing those sounds together to produce a desired effect.

Although there are professional sound designers that go to great lengths to record and prepare high quality speech and sounds, anyone can get involved with creating voice recordings and sound effects using free audio software and standard PC microphones to record dialogue, materials colliding, and sounds we can make with our voices.

I still use edited recordings of mouth sounds in my projects. The popping sounds made by colliding blocks in Topple were made with my lips, and the crackling whoosh flame sound effect in burnit was a mixture of crumpling paper and blowing into the microphone.

WHAT IS THE PROCESS TO MAKING VIDEO GAMES?

First I figure out what it is that I want to do. Do I intend to make a space shooter? A platformer? Is

my emphasis on expressive play, on story telling, or on action?

Then I determine what I have access to in order to create it, and how long I'll give myself for the project. Do I know someone who makes great animal animations, or realistic sounds, that would be willing to collaborate with me on the project? Will I schedule the game to be created in a few weeks, a few months, or longer? (In rare cases, as a stunt, I have made games in a few hours, although it shows in the quality and simplicity of what I produced during that time.)

I draw a picture of what the screen might look like when the game is being played. This leads to coming up with tentative answers to important questions, such as how large the player's avatar is on screen, what types of enemies or objects are in the level, and what information is displayed in the interface (health bars? magic meter? lives counter?).

Next I type a computer program that brings the elements of my picture to life. This is often very basic at first, for example getting arrow keys or the mouse cursor to move the (not yet animated) player character, and having enemies wander randomly.

Over many drafts – just like writing a paper for school – I refine the idea by making changes to how it works. If the player feels too slow, I'll adjust numbers in the program to boost the player's movement; if the enemies seem too unaware, I'll give attention to making them seem more intelligent (accounting for additional information, such as how to navigate between rooms without getting stuck). During this period of the game's development, I'll try adding many features, only a few of which will likely be kept for the final version.

Once I have established a clearer idea for how the gameplay will work, based on many exploratory attempts in the previous step, I plan out what art, sound, music,

writing, level design, and other information is needed that can realistically be created with the time remaining in the game's production schedule.

Much of the development time, at this point, goes into making and refining those art, sound, music, writing, level design, and other elements identified.

During and after making those pieces, the game needs to be tested thoroughly to ensure that it doesn't have bugs (program code causing the game to crash or behave in unintended ways) or significant design flaws (trivial ways to bypass playing the game as intended, such as one type of attack being overpowered). Fixes and changes are made according to the feedback from testing.

Lastly, I prepare the final information needed to share the game with the world: adding instructions, improving the title screen, taking screenshots, writing a concise description, and posting

the game to the web for others to play.

HOW MANY PEOPLE DOES IT TAKE TO MAKE A VIDEO GAME?

I've made many videogames working alone. One person can be enough for small games.

Most teams that I have worked with, creating small to medium-sized smartphone games or downloadable freeware PC games, involved 3-15 people. Companies that develop packaged retail games are more often at least a few dozen, sometimes even hundreds of people working together.

HOW IS THE SETTING CREATED?

Setting comes from collaboration between many different types of considerations. Art, writing, gameplay, and technology need to be accounted for. (Those roles are not necessarily one per person; a small team might have one or two people dividing up all of those interests, whereas a large team might have teams of people accounting for each perspective.)

The team members then create the pictures, 3D models, and/or programming to make that space functional within a computer.

Someone with sound editing experience will get involved as the setting takes shape, creating ambient sounds like crickets, rain, faraway explosions, or other audio that helps establish the setting.

For games with a complex environment, a tool is often made especially for creating levels in the game. Just like Paint or Photoshop can be used to draw and save pictures, the level creation tool is developed to “draw” and save a level by placing wall segments, enemies, and so on.

However, not all videogames have setting (or characters!). Tetris and Bejeweled are two very well known games that do not have settings or characters.

Videogames that resemble movies, or real spaces, by representing story within a

navigable area through humanoid characters are only one type of videogame.

ARE THE SETTINGS AND CHARACTERS BASED OFF OF REAL LIFE?

There's no right answer here – this depends upon the project and the developers involved.

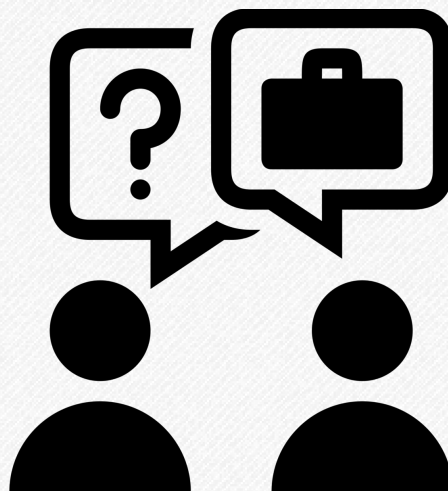
Some games focus on accurate depiction of real life as their appeal, trying to recreate the appearance and relative performance of various professional sports players or military figures in settings that are based closely on real life environments. On the other hand, games like Mario Brothers are pure fantasy, taking place with cartoony characters in silly settings.

There are also games that fall somewhere in-between, which try to establish a sense of authenticity by creating realistic, life-like environments and settings, even though the actual characters and places involved are imagined. This mixed approach is common with

war games; the war may have really happened, and the soldiers may be believable characters, but the actual locations and people depicted are made up in a way that fits what we know about related events in history.

CLASS QUESTIONS ABOUT GAME DEVELOPMENT CAREER

Part of what I love about videogame development is that it's exciting enough to help motivate people to overcome hard challenges. Making games well requires learning skills - abstract, practical, and even social.



Jon writes: *Hello. I have some questions I would like to ask you because my class is doing a project on what we would like to do when we get older. I chose video game design. I was wondering if you could answer my questions to help me with my interview? The questions I have are listed below.*

HOW LONG HAVE YOU BEEN DOING THIS?

I started programming games as a hobby starting around 1998 when I was in middle school. For a year or two before that I experimented a bit with modding commercial games (making levels, characters, and weapons for full games that I already played). I have made

games professionally off and on since 2005.

DO YOU MAINLY WORK WITH PEOPLE, DATA, THINGS, OR IDEAS?

In this line of work I begin by coming up with ideas, followed by working with those ideas to figure out how to turn them into data, after which I present that data to other people. The work a lot of my peers do in game development involves gathering a ton of data about how people interact with these things, leading to ideas

about what to change about their games.

Since the chain of events begins with ideas, in response to the question, I will pick ideas. Though not in the sense of “just” having ideas - a bunch of work with people, data, and things is required to test, develop, and deliver on those ideas.

WHAT DO YOU LIKE MOST ABOUT IT?

I like that each project is different, bringing new and different types of challenges. I also like the ability to “wear many different hats,” performing a variety of types of work on each project. I get to lay out levels, program exactly how things work, pull together the work of other talented people creating music and art, plan out and oversee the player’s experience (what they first see, how the menus look and work, what we show or give them for winning and doing well), etc.

WHAT ARE THE MOST FREQUENTLY RECURRING PROBLEMS?

Communication is a challenge every step of the way. When starting a project, there’s a challenge of communicating with other collaborators to make sure people understand what they’re getting involved with, since even a very talented developer will be unhappy with the results if working on a mismatched project. Even on a solo project, there are difficulties in what amounts to communicating clearly with myself, like sorting out and prioritizing my plans. During a game’s development, there’s another challenge in integrating the ideas of team members while providing sufficient communication to keep everyone on the same page. Further into a project, there’s a much different but very important challenge in trying to communicate with potential players regarding where they experience frustration or confusion during testing, and then

after release what the new game is like and about.

HOW DID YOU GET STARTED?

Although I began modding commercial games in late elementary school to get more out of single player games that I enjoyed on PC, I didn't really produce anything complete so much as use it as a context to practice digital image and audio editing, and thinking in detail about how games are put together. As a middle school student I taught myself the basics of C programming from books. I pretty quickly shifted from writing text programs into figuring out how to use my programming for simple games instead by displaying graphics and playing sounds. To do that at that time, I first copied assembly functions I found in the back of programming books. A little later came the Allegro game programming library, but there are now many other alternatives with various advantages and disadvantages. I

originally mostly made variations on classic games, since those were projects small enough to do well in a reasonable time frame while working alone. When I began my undergraduate years at college, I helped start a computer game development club, leading to larger and more involved team projects. I landed my first internship in the game industry through a recruiter that we invited to speak at one of our meetings.

WHAT TYPE OF TRAINING IS NEEDED TODAY?

A computer science degree can be very helpful, however people come into game development from a variety of backgrounds. Many people continue on to a master's degree in a videogame-related field to further help their qualifications to work in the game industry. All of that is primarily if you are looking to do it professionally, though. If you'd like to explore it first or primarily as a hobby, no formal training is necessary. Much of a modern game developer's training today is

finding ways to make (or mod) videogames on our own time.

WHAT TYPE OF TECHNOLOGY DO YOU USE?

I work on a home computer, no different from the sort used in many people's homes and schools.

In terms of software, I tend to use plain text editors to write code, and recompile using simple bash or batch scripts. That's a bit of an old-fashioned way to go about it. Many developers prefer managing their projects in more robust editors designed specifically for programming.

In regard to programming languages, I've done professional work in ActionScript 3 (compiles to Flash), C/C++, Objective-C (for iPhone), C#/JavaScript (for Unity), php with SQL, and a few scripting languages. I also use image editing software, like Photoshop or the free alternative at www.gimp.org, and sound editing/creation programs like Audacity.

WHAT CHANGES HAVE YOU SEEN IN THE PAST FEW YEARS THAT AFFECT THIS AS A CAREER PATH?

Every few years, a different business model appears that draws the attention of investors and businesspeople. In order to work professionally, game developers have to find ways to design projects that work well with those new business models. 30 years ago, it was paying a quarter to play at the arcades. For the next 10-20 years, it was largely selling games as \$60 retail goods, and then the MMO/subscription model seemed to dominate for awhile, followed by cheap \$0.99 mobile games. More recently the free-to-play micro-transaction model used by social games seems to have taken over. Old business models remain of course – there are still new arcade games, and a major part of the industry continues to be \$60 console games sold retail – but the arrival of new, larger market segments makes those an increasingly small percentage of professional game development.

What I can say for sure is that this space is sure to continue evolving and changing, so it's important for a videogame developer to be ready to frequently adapt and always be learning new techniques, platforms, and audience considerations.

WHAT PERSONAL QUALITIES DO YOU FEEL ARE NEEDED TO SUCCEED IN THIS?

Like many cutting edge fields, success in game development relies upon determination, persistence, patience, and willingness to self-educate and self-direct. Many skills are needed that go beyond and build upon what anyone currently teaches, and the only way to pick up those skills is through experimentation, working through problems that arise, connecting with peers working on the same challenges, and getting a lot of practice.

Another personal quality that helps is knowing how to be flexible while still finishing the work. Rather than stubbornly sticking to something until it's exactly the way originally

intended, in many cases being a good listener and open-minded observer can save or productively redirect work by noticing what parts of the task matter most. Ideally this leads to finding a clever way to achieve those goals without getting stuck on the many details that may turn out to be much less important, or even distracting.

Lastly, but importantly, it will go a long way to learn how to present well. Seek opportunities to practice presenting your results, your mid-process progress, and even presenting yourself (as a potential collaborator, as the right person for a particular task, etc.). Think about any game that you enjoy – if someone hadn't figured out the right way to show that end result to strangers, and to win others in their company or team into collaborating on it, we could never have had the opportunity to play it.

When you begin making nifty projects that you're proud of, don't

hide them from the world. Be prepared to show and share them in an appropriate way.

WHAT ADVICE COULD YOU GIVE TO ME WHILE I AM PURSUING THIS?

Game design involves doing a lot of different kinds of tasks, working with a lot of different kinds of information, and communicating with a lot of different kinds of people. Working to become well-rounded is important in this.

Technical understanding is useful (math and science), creative practice is useful (English and art), and the other random information someone knows about (history, literature, film) is what separates their ideas and work from that done by other equally capable people. Even clubs and sports can provide useful insights, habits, and ways to relate that might be missed otherwise.

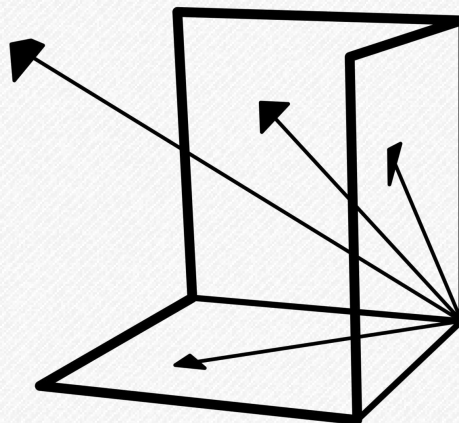
There were times in school when my friends wouldn't see the point of a geometry assignment, but I would be excited about it because I knew how I could use it to make

my videogames do things I didn't previously understand how to do. Writing classes, which didn't seem so obviously applicable to me at the time, have turned out to be every bit as important, as I use much of the same process and ways of thinking now for my non-writing work.

In other words: despite my previous point being the importance of teaching yourself things that other people can't or don't yet teach, this point is the importance of paying attention and learning from others about the material that others can and do teach. Self-education is often the only way to learn certain things worth knowing, but when it comes to material that people have spent thousands of years finding ways to explain, don't insist on reinventing the wheel and trying to rediscover all that yourself.

MATH FOR VIDEOGAME MAKING (OR: WILL I USE CALCULUS?)

Math in school tends to be very abstract, and involves memorizing a lot of rules. For games the math is much more clearly situated in solving real, often simple problems. And you can look up past solutions any time.



Q: How important do you think it is to learn calculus for game programming? Which math fields are most useful in hobby game development?

A: I needed calculus for classwork. I have virtually never used it for any of my dozens of freeware and small commercial projects.

That said, I'm not very involved in writing physics or 3D graphics libraries – when I need those features, I reference libraries written by other people who are more knowledgeable in those areas. Someone working on

rendering features or optimizations for 3D engines might well have more use for calculus than I do.

For most gameplay-related math – jumping, bumping into things, the speed of a shield recharge – simplified approximation suffices. In many cases the sloppier estimation approach is even better than trying to work out proper precision. As one example, the tried-and-true platform game mechanic of “how long the jump button is held after leaving the ground determines jump height”

clearly does not in any way reflect the real physics of jumping. It's more important for a videogame to behave as the player wants (or expects) than it is for a videogame to behave realistically.

Of course, if we were trying to design an authentic flight simulator for military training, or model a car accident for the purposes of studying safety and real-world engineering, this would be a very different story. But for games, we can usually just twiddle with numbers until things look and feel right. Videogame spaceships and helicopters move because we add a velocity value to their coordinates every frame – not because of rocket propulsion or Bernoulli's principle. For 2D game programming, spacecrafts and helicopters only need 1st grade arithmetic.

But beyond basic gameplay interactions, here are the fields of math that I run into most frequently for game programming,

along with notes on what I find each most useful for:

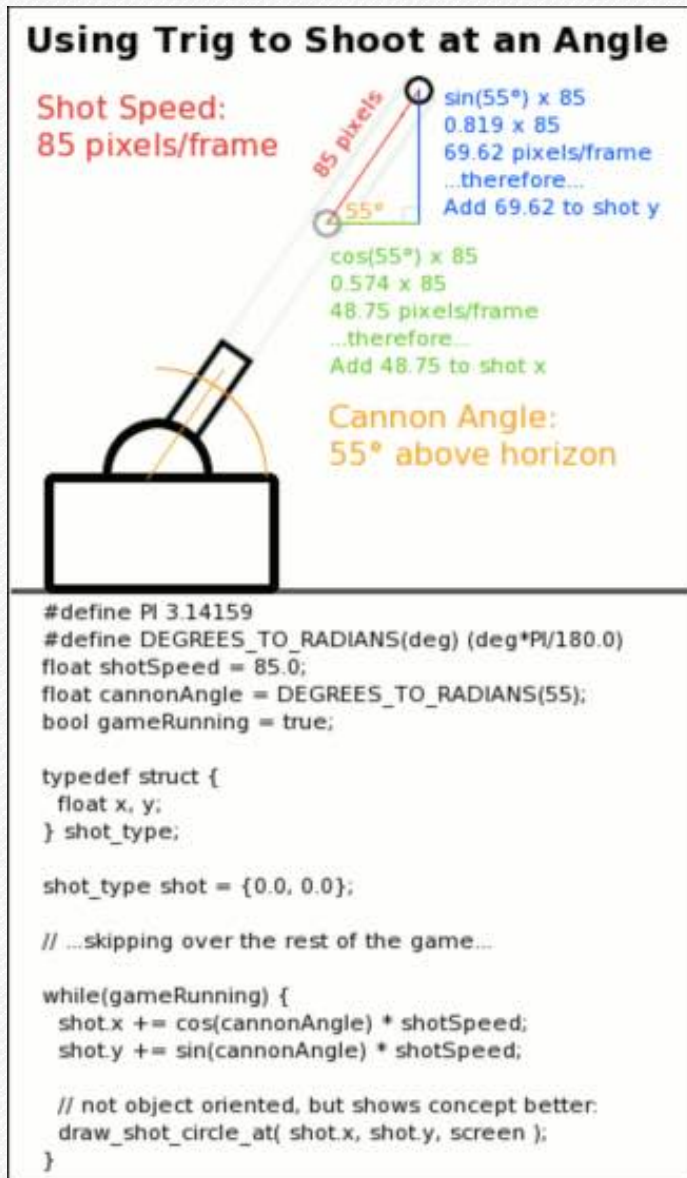
BASIC GEOMETRY

2D graphics programming is a matter of nudging object coordinates around on a plane, then drawing images every frame based on those coordinates. Bounding boxes (comparing coordinate differences) and the distance formula (Pythagorean Theorem) are commonly used to test whether two objects have collided with one another.

TRIGONOMETRY.

Sine and cosine are helpful for angle-to-component translation, for example when determining what percentage of a bullet's total velocity the x-speed and y-speed are given the angle of a firing cannon (see illustration on the next page) – this also applies to an overhead race car. Atan2 is handy for getting an angle between two things, given their relative offset in grid locations – good for getting an enemy to face the player,

pointing a simple homing missile, and so on.



This is why and how I learned simple trigonometry. I needed a way to get my virtual anti-aircraft gun in Sky Rake (1997) to fire a projectile at the same speed independent of which angle it's fired in.

VECTOR GEOMETRY

The dot product is an incredibly versatile math operation to gain mastery of, even in the simplest 2 vector 2D case. Learn how to use

it, and you'll greatly increase the number of nifty things that you can pull off in game programming. This is useful for reflection off angled surfaces, checking line-of-sight, determining which direction something is facing, and many other common spatial/angle relationships.

Normalizing vectors is another basic and useful aspect of vector geometry. Properly applied, these offer a more efficient way to accomplish some of the same things that trigonometry is commonly used for, including aggressive homing missile logic.

BOOLEAN LOGIC

It's common in game code to want something to only happen either when multiple things are true (if player is on ground AND pressing jump button, then jump) or when at least one of several things is true (if [W Key] OR [Up Arrow] is pressed, then try to jump). At first glance this doesn't look like math, but when these statements begin to compound, there's are math-

like rules that can be used to untangle and simplify boolean logic.

SIMPLE ALGEBRA

Old-fashioned line-intersection check calculations are great, and trivial to write as a function. This comes up for things like pong AI, which needs to anticipate where an elastically bouncing ball will intersect a screen edge, or finding if and where a laser or bullet shot hits a wall.

MODULAR ARITHMETIC

The modulus (signified by the % sign in many programming languages) returns the remainder after dividing one number by another. This can be used to bound a number within a given range, such as ensuring that a potentially huge positive number points to a valid array entry, or limiting a random number to being less than the divisor given. Example: `rand()%35` is a common way to get a random integer from 0-34, inclusive.

MATRIX ALGEBRA

In 3D, matrix math is used to perform faster coordinate translations and rotations to render models at various locations and angles. In 2D, rotating a Tetris block 90 degrees involves a little matrix math.

TILE ARITHMETIC

Admittedly, this isn't really a math field. But it is very important for game programmers. I'm referring to the use of multiplication to translate grid coordinates from a 2D array to world pixel/character coordinates, and division to match pixel/character coordinates to corresponding indexes in a 2D array. Turning a pixel coordinate (208 pixels from the left edge of the world) into a tile index (assuming 16 pixel tiles, $208/16 = 13$, so the 13th tile) gives a fast way to check what type of tile fills that location in a 2D array. If the player is found to be standing on a wall tile, the player can be bumped backward; if a brick breaker ball hits a brick tile, it can change the value at that tile's array

index to clear it, and reflect the ball's movement.

Basic Tile Math - RPG and Brick Breaker Examples



```
#define T_WIDTH 16
#define T_HEIGHT 16

#define TILES_W 7
#define TILES_H 5

enum { TILE_FLOOR, // 0
       TILE_WALL, // 1
       TILE_DOOR, // 2
       TILE_KEY, // 3
       TILE_START; // 4

int world[TILES_H][TILES_W] =
{
  {1,1,1,1,1,1,1},
  {1,1,1,1,3,1,1},
  {1,0,4,0,0,0,2},
  {1,1,0,0,0,0,1},
  {1,1,1,2,1,1,1};

// ...skip most of game...
for(int h=0;h<TILES_H;h++)
{
  for(int w=0;w<TILES_W;w++)
  {
    // assuming a function
    // that draws a given tile
    // type to a given x/y pixel
    draw_tile( world[h][w],
               w * T_WIDTH,
               h * T_HEIGHT);
  }
}

// .. skip more code...
int pTileX = playerX / T_WIDTH;
int pTileY = playerY / T_HEIGHT;

switch( world[pTileY][pTileX]) {
  // ...player on wall? key? door?
}
```



```
#define B_WIDTH 32
#define B_HEIGHT 16

#define BRICKS_W 5
#define BRICKS_H 5

enum { BRICK_NONE, // 0
       BRICK_PLAIN, // 1
       BRICK_DOUBLE, // 2
       BRICK_NEVERBREAK; // 3

int wall[BRICKS_H][BRICKS_W] =
{
  {2,1,1,1,2},
  {2,1,1,1,2},
  {3,2,1,0,3},
  {1,1,0,0,1},
  {1,0,0,0,0};

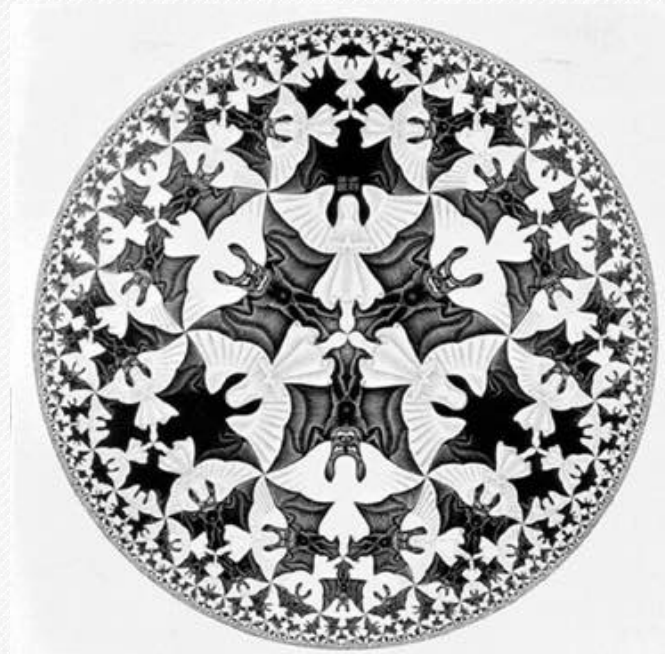
// ... skipping game code...
for(int h=0;h<BRICKS_H;h++)
{
  for(int w=0;w<BRICKS_W;w++)
  {
    // assuming a function
    // that draws a given brick
    // type to a given x/y pixel
    draw_brick( wall[h][w],
                w * B_WIDTH,
                h * B_HEIGHT);
  }
}

// .. skip more code...
int bTileX = ballX / B_WIDTH;
int bTileY = ballY / B_HEIGHT;

switch( wall[bTileY][bTileX]) {
  /* ...what brick type is the ball
  overlapping this frame? */
}
```

Tile-based arithmetic for world collision calculations is a common memory, performance, and level design optimization used in many historical game genres.

Another important thing to keep in mind, pertinent to any sort of math used for videogame development,



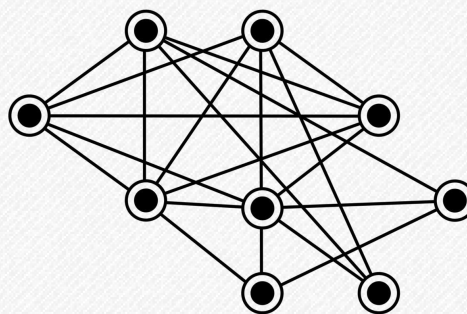
M.C. Escher, known for carving crazy wood prints in the 1930's that we'd have trouble tricking a modern computer into rendering, was not a mathematical generalist. He didn't learn everything that he could find to learn about math, only to use whatever tiny fraction of it applied to his work. Quite the opposite – he dabbled in just as much math as he needed to create what interested him.

is that in good programming, every problem is only solved once. That solution is then abstracted into a reusable function.

Frequently, then, math needed for game programming can be looked up on an as-needed basis, coded into a new semi-general function that accepts usage parameters, then largely forgotten, freeing you to think instead about what it does functionally for your game.

QUESTION ABOUT COMP SCI AND GAME DEVELOPMENT

Though Computer Science is very powerful to know, it's not the same as game programming. Most C.S. isn't videogame programming, and vice versa. This may be a relief to those that do not study Computer Science.



Question, Part 1

Hi Chris. This is my first year in college, and I'm currently a Computer Science [CS] major.

However I came in with no prior programming experience, so the math and programming classes so far have been pretty tough. I am thinking about switching my major to something more enjoyable, and just trying to teach my self game development instead. Any thoughts on whether, or maybe why, it would help to stick it out with CS? My dream job is to work in the gaming industry, though I may wind up looking into indie development.

Answer, Part 1

First, full disclosure: I was a CS major, too. That may bias my view of it, but it also means that I can provide an informed, first-hand look at how it did or didn't help.

Everyone's path in life and career is different. The decision is purely up to you, and as you necessarily know more about yourself than anyone else does, it's important to weigh my thoughts on this as just one more source of data.

Whether you decide to seek corporate industry work or go independent, I think that a Computer Science degree is still among the most applicable, useful, and broadly marketable backgrounds related to game development. I'll do my best here to make a case for why I think that's true.

Note too that, importantly, CS is also a credential respected outside of games, which can make it a practical degree to have for finding or creating back-up plans down the line if necessary.

COMP SCIENCE IS NOT GAME MAKING

As you've probably already discovered, game development generally isn't really part of CS curriculum, except perhaps for a single high level elective class. Almost all of my game development during college came from extracurricular clubs.

Learning game development requires a good deal of self-teaching no matter what major someone is undertaking.

Experience in CS is relevant though, and it complements someone's self-education in game development by pushing development in areas that it would be easy to overlook on our own. Material from CS can be useful outside of the classroom, when working through real-world game development problems, sometimes in ways that are hard to recognize until they come up during the creative process as either challenges you've prepared for, or – if the skills aren't there – as apparent dead-ends.

There are no doubt routes to take that are easier or more enjoyable to take in the short term, but those paths may or may not be easier and more enjoyable in the long term. College years are partly about investing in your future capabilities and opportunities; the standards and challenges of a CS degree are just one of the established ways to stay on a solid track to do just that.

If it's not immediately clicking, it might require spending some more time in the library studying (even if you don't need the books there for CS, it's handy as a well-lit place filled with other people calmly working, too), attending more of the optional tutoring and office-hours sessions available through your school, starting earlier on assignments, and staying up later sometimes to hack through getting assignments figured out and working. Many college students wind up automatically paying for resources but not making full use of them, whether via the career center, library assistance, or office hours. Before deciding that a subject isn't working out, make sure that you're at least getting your money's worth and using all the mechanisms available. Outside of working on videogame projects and trying to live a rounded life in college, I spent a ton of time at office hours asking questions to try to fill in for what I didn't

understand yet from the lectures and reading.

BIG COMPANY VS INDIE

A qualifier about the difference between big company work and an indie career: earnings from independent development can be really, really hit or miss, spotty and unpredictable, and in the majority of cases unprofitable. There's a huge selection bias in the indies we hear the most about – they're disproportionately the rare success stories that have done extraordinarily well. The countless others that are barely scraping by or losing money don't make for good headlines.

Independent work can be a tricky and trying path to navigate, and though it's potentially very rewarding, it should perhaps be initially thought of as something to develop on the side of some more consistent line of work as the main plan. With a bit of momentum from giving yourself a head start, and a bit of savings in the bank from doing something with a more

predictable paycheck for awhile first, you'll have better odds of lifting off the runway.

One element that helped me in my journey – though again each individual's mileage from such choices will vary – was to minor in Business Administration. Those courses involved learning and practicing skills that were useful in every scale of work environment that I've been involved with, including my time alone: presenting information, ideas, yourself to others, basic financial concepts, project planning, even practical bits like resume preparation. (Even if you go independent, keeping an efficient, up-to-date, professionally presented summary of skills and work can be useful to have on hand.)

Whatever you decide, good luck on the road ahead!

Question, Part 2

It seems like many of today's big designers got into the industry with a non-technical major.

Do you see the industry moving more towards requiring a relatable degree?

Answer, Part 2

You're right that many of the current big and/or historical names in the industry have a seemingly random assortment of backgrounds. John Romero and David Perry were both, if I recall, completely self-taught, and did not attend university. Nintendo's Shigeru Miyamoto went to school for industrial design. In my interview with Atari Adventure designer Warren Robinett, he mentioned that one of his early co-workers in the game industry, "had a degree in Zoology."

Of course, Shigeru Miyamoto was born in 1952 – so for perspective, Pong hit the market (overseas, here in the US) when he was already 20 years old. Robinett's undergraduate degree was not in Computer Science – he majored in a CS-like area of math, but when he was an undergrad CS had not yet become a standardized field of study.

That historical difference partly accounts for what you've rightly pointed out.

40 YEARS AGO

The industry didn't exist when most of the early innovators were growing up. To the extent that some degrees were later relatable, like Robinett's pre-CS technical math major, no one at that time could have picked it for that purpose, because those exact jobs and products didn't exist yet.

Another thing to keep in mind was that computing was very different in the 1960's to early 1970's. Far fewer people had access to computers, but among those that did, a much higher percentage of users taught themselves how to program because it was pretty central to being able to use a computer at all. Programming was one of the few activities that could be done on machines at the time; the first word processor didn't come out until 1972, and the internet didn't become common

and commercialized until much later in the 90's.

There also wasn't much 'prior art' for a game developer to catch up on. Nearly everything being done at all was new, and by casual assessment equally valid from a business perspective until the market response to a shipped product indicated otherwise. By comparison, much has since been sorted out by now, both from a technical perspective and also from a business perspective.

People have already lost a lot of time and money figuring out how to do certain things in impressively efficient ways, how to collaborate effectively on team projects, and discovering what consumers responded to well or poorly. Developers caught up on those findings have an advantage over others starting from scratch, reinventing the wheel, insisting on learning the hard way.

Now that four decades have passed, many of the surviving companies have thousands of

employees (or at least compete against companies of that size), and there's demand for high degrees of specialization for a team to be able to distinguish its products from those that competitors are able to create. Even just keeping up with inflated consumer expectations, especially on today's higher fidelity platforms, can require specialists. Whether applying for an industry job or creating independent games as a small team, each individual is competing against a tidal wave of other eager, passionate people that have partly designed their education and adult lives around developing skills that have been found relevant to the now more established craft of videogame production.

INCREASED IMPORTANCE OF DEGREES

The increased importance of degrees, though more so for applying to jobs as opposed to working independently, has also become a useful shortcut for hiring managers serving as a

company's first line of assessment. At a company with a recognizable brand, the executive producer or lead engineer can't personally screen every applicant. Instead, the initial set of applicant submissions first has to make it past recruiters that are not necessarily game development specialists, and thus may look for specific degrees and types of prior experience as evidence of the skills they're hiring for.

Applications making that cut then move on to one or more rounds of interviews, at which point an appropriate specialist from the team may be available to help assess actual qualifications, but getting to that phase with little or no relevant professional experience is part of where the appropriate degree can help.

While there may be capable applicants without relevant degrees, unfortunately there are often enough qualified applicants that do have relevant degrees that it's easier and more cost and time

efficient for a company to hire primarily from the latter pool, for which less screening may be required since college admissions and professors may be thought of as a very specific sort of filter.

COMP SCI IS MORE THAN PROGRAMMING

A CS degree is not just about knowing how to program. Having a particular degree is often also interpreted as evidence, correlative beyond the scope of what's covered directly by the curriculum for that degree, that someone is dedicating themselves to some predictable set of knowledge, skills, and values. Companies need all kinds of people to thrive, but from a purely practical perspective, it doesn't take much imagination to picture how a "CS person" can fit in and benefit a company that creates software. Computer Science rewards attention to efficiency, feasibility, robustness, extensibility, self-correction, and some other less easily pinpointed but equally useful priorities, in ways that

specifically apply to computer programs. (i.e. while it's true that plenty of subjects may reward attention to efficiency and feasibility, traditional CS approaches these in an extremely rigorous fashion by dealing with their theoretical limits.)

The increase in people hired that have relevant degrees, from the above factors, has created a positive feedback loop. When the people with those credentials get promoted over the years into more senior-level positions, they're even more likely to value that same (or similar) credential among incoming candidates at an interview stage.

GAME DEGREES

Another option within the spectrum – which you haven't asked about directly but I'll address here for sake of completeness – is in degrees more specifically and narrowly about videogame development. These haven't been around as long as Computer Science, making them a bit less standardized, and

consequently more of an unknown, unproven value to many people in industry. Anyone with a CS background has a pretty clear idea of what someone with a CS degree from another school likely covered, but there's so much variety between game degrees that it's hard to know what skills and work culture to expect unless someone in a hiring position is personally familiar with that exact school and educational program.

Partly as an effect of having not been around as long, there are also not many senior-level people (yet) from those types of educational backgrounds.

The flip side is that some of the same factors that have made the CS degree of more interest to the industry are likely at work for those more game specific degrees: easy shortcut for hiring managers, evidence of (probable) commitment to certain knowledge and attitudes beyond the curriculum, and as time progresses, more senior level

people coming from that background will be held up as evidence of the type of value associated with those degrees.

That path can also be a more risky value proposition to students, however, since those degrees sometimes cost comparatively more (being seen as more vocational, and thus framed more as a financial investment) and can be more limiting to videogame industry work. Students with those degrees have less of a clear back-up plan, not only if they can't find a fit in the game industry, but also if the game industry undergoes a major shift as it did when arcades mostly phased out, when MMO's seemed like the next or only big thing for awhile, when digital mobile distribution lowered barriers-to-entry for competitors with much smaller staff, and as social games have grown to absorb an increasingly sizable chunk of game company investment capital.

The value of a mostly game-specific degree as a back-up plan varies greatly between how individual schools have chosen to implement such programs. I have heard peers in such programs report everything from their studies being typical Computer Science in-disguise, to specific tool training (which can seem the most useful in the short term, but can be the least useful in the long term – teach yourself the tools, instead!), to amounting to little more than an excuse for networking. Remember though that a recruiter without direct connections to the particular program (perhaps less likely outside the industry than within it) may not have an easy way to quickly discern one of the above from another, and that could affect whether an application makes it to a stage in the process at which someone with the necessary domain knowledge takes the time to sort out the candidate's relevant capabilities.

CLOSING

The case, by contrast, for a CS degree is that its emphasis is on fundamental concepts, which aim to extend well into the future largely independent of specific technologies or how the market may change. At any time in game industry history, even just a five year span has covered some pretty dramatic changes, but a proficiency for the skills and mindset of proper software development has remained central and relevant.

Lastly, as a reminder: I have only an outside, second-hand perspective of the alternatives. Your best bet for gaining a more balanced perspective would be to pitch this same question to some game developers with different backgrounds. There's no one right way to go about it, though the odds are better some ways than others.

THE WAYS OF SELF-EDUCATION

The world has never had so many resources to learn from, and our access to them with a simple internet connection is incredible! Yet for it to benefit us, we've got to make sense of what to find and how to use it.



YOU CAN TEACH YOURSELF

Many of the skills that you'll need to make videogames, you can teach yourself.

...says the guy that devotes much of his time to teaching videogame making.

However I'm talking here mainly about baseline skills. By providing this guide to navigating materials to help you teach yourself, I'm growing the number of developers that I can work with on the deeper, more interesting challenges that I find more engaging.

THERE'S A PROCESS TO IT

Being good with computers isn't about knowing everything. It's about knowing how to look up or figure out what you don't know. Similarly, teaching yourself to make videogames is largely a matter of knowing which resources are available to you, and how they can best be used to accelerate your progress.

There are a few other resources than Google to consider in this case, though. The main types of resources available to you for

teaching yourself basics are Books, Websites, Programming Language and API Docs, Forums and Peers, Code Libraries, Sample Code, Practice, and Past Project Files.

BOOKS

There are books on every topic, for every ability.

Many are available online for free, although a number of those may be relatively outdated.

Relying early on too much on purely online resources encourages sloppy habits that shortcut learning, like copying and pasting blocks of code instead of getting in the habit of typing it. Web resources are awesome for looking up information – they can be searched trivially if you know what you’re looking for – but to develop a full understanding of fundamental skills and a sequence of material can be really helpful.

Cost can seem like a factor, since good books on this subject can easily be in the \$30-\$80 range, but

the important things to think about are that (a.) the cost per hour is quite low, (b.) the overall cost pales in comparison to paying for classes, and (c.) it’s impossible to estimate value of “this professionally prepared and edited content was what enabled me to succeed in learning this” (I can say that last one about at least a few books I had growing up).

WEBSITES

Websites are terrific resources for providing answers to specific questions, and finding links to other useful resources like sample code. As explained above, I believe that they’re not as good for conveying fundamentals.

PROGRAMMING LANGUAGE AND API DOCUMENTATION

APIs, or Application Programming Interfaces, are massive branching directories of information indicating what is built into a programming language’s default functionality. Languages that are newer than C and C++, especially programming languages used for online content, have extensive

APIs, like the Java API (Java is great for learning programming, and though often not ideal as a final platform for intermediate/advanced game making, its API is a great example) and the ActionScript 3 API (still used for many web games).

There's more information in there than any one human being should read in their lifetime, and a good 95% of it probably isn't at all relevant to what you'll be using most of the time to make videogames. These are handy references to keep a link to, and can be a lifesaver in clearing up detailed questions about what to expect in terms of handling of errors and edge cases. But once again, digital resources are a good way to hunt for specific answers to specific questions you have, or to find other resources.

But learning to program starting from the API would be like trying to learn creative writing in a foreign language beginning with their native dictionary.

FORUMS AND PEERS

Just like in school, if you have a question, there's a pretty good chance that other people trying to learn the subject have this question, too.

Unlike in school, your class size ranks in the hundreds of thousands, and there's a detailed record of virtually every question asked in the past decade or longer.

If you're finding yourself stuck on something – and I know this seems obvious but I'd be remiss to not reiterate it – do an internet search or crawl relevant web forums for an answer. If you can't find one, then – keeping to the school analogy – do a favor to hundreds of thousands of people over the next decade by speaking up on forums with your question someplace that others with your same question will find the answer.

CODE LIBRARIES

In programming, a library is just a packaged chunk of code someone

else wrote to handle specific functionality. This might include network operations, text display, sound playing, image manipulation, loading, displaying, physics simulation, etc.

A good programming library can address a problem that you had no clue how to solve, in a way that's (a.) fast (b.) easy (c.) leaves you clueless how to solve it without using the library. If your goal is learning about something in particular, like graphics programming or network code, point (c) may be a problem. In many cases, a library is used to do something that very smart people have spent decades researching and perfecting. In that case, that library can save you literally years of frustration, and let you focus instead on what you're doing with the graphics, network, and physics code to make your game stand out.

SAMPLE CODE
Jackpot!

Finding good sample code can be the difference between taking hopeless shots in the dark about how things tie together, and working with confidence that what you're starting with should work. This is also invaluable early on because by starting with code that you know should work, it can help you determine whether your programming environment is installed and configured properly – if it fails with sample code that came from a dependable source, you know that your project, environment, or compiler isn't set up correctly. That's way better than tearing your hair out hopelessly changing things in your code wondering why it's not working.

Find sample code. Read sample code. Compile sample code. Retype sample code. Modify sample code. Try commenting out various lines of sample code to see how it affects what happens. Add comments to sample code.

Search for sample code appropriate for your level of comprehension. If it's not too long, and it doesn't make perfect sense to you just yet, verify that it compiles in your environment, then print a copy to keep folded in one of your books for occasional review.

PRACTICE

Good programmers are dramatically faster and more efficient than less good programmers.

“Good” here isn't in reference to total years of experience, how many different programming languages someone knows, or what projects someone has been a part of building. “Good” is referring to how quickly someone can synthesize the code needed to solve a problem, how well that someone can generate code which is structured with future needs, readability, and adaptation in mind, and how easily the person can digest and interpret code put in front of them.

Some people are as fluent and comfortable with programming as they are with their primary speaking language. Others fumble through it like it's a secondary language they remember traces of from when they took a few semesters of it many years ago in school.

I chalk up that difference to the comfort that comes from countless hours of practice. The kind of usage that goes beyond fulfilling a specification someone else handed over. The kind of experience that comes from stretching and applying knowledge to solve problem or create a project that the programmer was personally invested in.

On one end of the spectrum, there's the way that someone with zero background in programming, but access to the internet, could cobble together code they don't fully understand to make something happen. On the other end of the spectrum, there's

someone that can just breeze through whatever problem you put in front of them, perhaps occasionally taking a moment to reference documentation or copy something out of their vast library of past project files.

I'm reiterating this distinction for an important reason:

1.) You absolutely, positively, unmistakably, want to be in that latter group. You want to be the person that is really on top of their stuff.

2.) There is no doubt, none whatsoever, about why you will or won't wind up in that latter group, instead of that former group: practice.

YOUR PAST PROJECT FILES

Your past project files are much like sample code. The main differences are that you understand everything in it.

Because you were responsible for the whys and the hows, you know exactly where to find what.

Time and time again, my mountain of past project code has been the source of my confidence in taking on a new project. Provided that I've done something in a past project, or done something like it, there's no doubt in my mind that I know how to do it, and can probably make it work again without much fumbling around. Instead of spending my implementation time thinking about how to do something, I could start from my past solution, and spend that time finding a way to do that something better.

The only way to get that mountain of past project code is to (here it is again!) practice. On your own time.

If it's done to satisfy classwork requirements, it won't be any different than what every other programmer has been exposed to.

If it's done to satisfy business goals, it's likely company property that you're not free to reuse.

Practice. On your own time.

3



Programming is the language of the machine. No skill will serve as a more general foundation in empowering you to see through your own unique ideas. Even if you want to be a level designer or artist, you will be able to create the context and tools for your work by coding.

PROGRAMMING

GAME PROGRAMMING FUNDAMENTALS

Many people getting into game programming are first learning - or have already learned - some common programming elements. Here's how those generic pieces tend to show up in videogame making.



Programming Basics in Game Context

The goal of this section is certainly not to immediately transform someone with no prior programming experience into a game programmer. That takes time and practice, reading and trying, experimenting and thinking, over a period of weeks (basics), months (intermediate), and years (expert). This is not intended to take the place of a good book on programming, or practice – it's here for warm up, review, or to

complement other resources that are out there by providing a connection to how the most common code structures are used in videogame development.

My hope is that I can plant seeds of familiarity, and establish context of use for the various elements of programming.

The following examples are given mostly in C code, taking advantage of a little bit of the flexibility given by a C++ compiler. C++, Objective-C, ActionScript 3,

and Java use the same or nearly identical structures for the same functionality, and those are the major programming languages used to develop today's console, downloadable, mobile, and web videogames.

Finally, the text in this section introduces a lot of material, in a very short space for the amount presented. If you're new to programming, it will take more than one read, but there's a good chance that you'll pick up something new each time.

Remember: programming is how every videogame on the market (Wii, online, mobile, 360, PS3, SNES, Atari, Arcade...), every piece of software you use (Office, Windows, FireFox...), and every electronic device works. There's some patience involved in understanding how all these things work. Stick with it – you'll be glad you did!

Whitespace

WHAT IT IS:

Whitespace refers to the gaps between letters within program code, whether represented as tabs, spaces, or new-line breaks (as from the Enter key). Many modern programming languages treat all whitespace the same, meaning that whether you skip lines, indent a certain distance, put two spaces instead of one, or use tabs instead of spaces, the program code will work exactly the same way.

HOW IT LOOKS:

```
for(int i=0;i<5;i++){showNumber(i);doOtherThingToo();}
```

...works the same as, but is not as easy on human eyes as...

```
for(int i = 0; i < 5; i++) {  
    showNumber(i);  
    doOtherThingToo();  
}
```

EXAMPLE OF A USE IN VIDEOGAMES:

When a program's source code doesn't compile correctly, especially for a beginning programmer, it's frequently due to a mismatch between braces (the { and } symbols). Effective use of indentation to indicate how many

pairs of braces are around code helps keep track of where braces should close. All errors in programming code are reported by the compiler using line numbers, which are only useful in leading the programmer to the problem if the code is broken onto new lines after each individual instruction is ended by its semicolon.

The Comment

WHAT IT IS:

Text visible only to the human reader, that the computer ignores when it's time to generate ("compile") a program from the program code.

HOW IT LOOKS:

```
// Text to the right of two slashes is a comment  
  
/* ...and so is anything typed between the slash-  
star and the star slash. This type of comment,  
unlike the double-slash style, works on multiple  
lines. */
```

EXAMPLE OF A USE IN VIDEOGAMES:

If you program something at the start of a project, then return to that part of code weeks or months later, you may have trouble figuring out what you were intending when the original code

was written. Does the collision detection algorithm assume two objects weren't already overlapping in the previous frame? Is there a limit to how many bad guys the game's graphics code is intended to render at once?

Comments are a perfect way to record this thinking alongside the code to which it's relevant to ensure that it's passed along to your future self, or anyone else that might wind up reading the code.

Names

WHAT IT IS:

It is up to the programmer to create names for most things used in a program's code – names to keep track of numbers that are being used differently, to refer to text phrases that serve different purposes, and names for different chunks of code to succinctly describe their functionality. The computer doesn't care what these names mean to people ("drawStuff", "ElvisPants", and "BV3_aUiP0" are all equally valid),

but their consistent usage in different parts of the program is how the programmer stores, changes, and checks values for different purposes.

Even though the computer can't tell the difference between a function for updating the screen called "UpdateScreen" and one called "DogsGetExcited," the former is a smarter choice because it reflects how the programmer is using the label. Note that whatever names you use inside your program are 100% invisible to the users and the outside world – they're just a way to stay more organized about the numbers you're shuffling around.

HOW IT LOOKS:

```
moveBadguys(); // good name for a code chunk
int playerPositionX; // clear name for a number
abc123(); // terribly unclear name. What's it do?
int r; // bad! radius? red? radiation? roundness?
```

EXAMPLE OF A USE IN VIDEOGAMES:

If you're good with naming all the numbers, code sections, files, and other labels needed by your program, the code can become impressively clear to anyone

browsing through it. That means less time is needed for writing extensive comments to explain exceptions or clarify meaning, and far less time will get drained into untangling confusion (or problems arising from confusion) based in poor name choices.

Sequence

WHAT IT IS:

With only very rare exceptions (ex. very advanced topics related to PS3, XBox 360, and high-end computer programming), program code executes only 1 instruction at a time, 1 step at a time, top-to-bottom just like English text is read. Every instruction goes on its own line, for sake of human readability, and must end with a semicolon, for sake of machine readability.

HOW IT LOOKS:

```
doThing(); // happens first
doOtherThing(); // then this
lastThingHappening(); // happens last
```

EXAMPLE OF A USE IN VIDEOGAMES:

```
moveByPlayerInput();
detectCollisions();
drawEverythingToScreen();
```


...is a smarter order than...

```
detectCollisions();  
moveByPlayerInput();  
drawEverythingToScreen();
```

...because in the second ordering, character positions aren't updated to account for collisions that could have happened by the player's input until after the current frame is drawn to the screen. That means the player may wind up seeing things overlapping one another, which could have been easily avoided by simply switching two lines of code!

Functions

WHAT IT IS:

A function is a mostly self-contained solution to a recurring problem. In its simplest form, it's simply a group of code that can be called from any other place in code, any number of times, with a one word label. More involved functions can accept input values for use in calculation, and/or can output new values reflecting the results of its processing. The parenthesis pair always show up after a function name when it's

called; if it is a function that takes input values, those values would be specified between the parenthesis.

Programming languages tend to have one initial entry function that always gets called: either `main()` in C/C++, or with a name matching the main file, class, or program in Java/ActionScript. Code begins working from the top of that function, and keeps going until it reaches the bottom of it, getting sidetracked into functions called from it.

HOW IT LOOKS:

The "declaration" takes a form like this:

```
void doThing() { // function with no input/output  
    // Code goes here... to do anything  
  
    // every time the function is called  
    // this code will execute, top-to-bottom,  
    // from the function's starting brace "{"  
    // to the function's ending brace "}"  
}
```

The "call" (use) often looks like this:

```
doThing(); // does "code goes here" from above  
doThing(); // runs that code again, a second time
```

EXAMPLE OF A USE IN VIDEOGAMES:

Grouping related code together into functions helps organize code

into more readable sections. A function called `drawStuff()` might contain calls to all other draw functions, such as those used for drawing the background, player, enemies, powerups, weapon projectiles, and health information, which in turn could call other functions that handle the details of how to display the necessary images in the necessary places. This approach enables you to separately take a detailed or big picture look at different parts of the program, breaking the functionality up into understandable chunks. It also centralizes code that's needed multiple times into one place, where it can be fixed or updated only once if something is found to be broken in its behavior.

In real-time videogames, there is generally one primary function that is set up to be called about 30 times per second, executing code in it from top to bottom. That function is expected to call other functions to update positions,

check for user input, and redraw the display – anything that needs to happen constantly. Common names for this function, by convention, are `draw()`, `update()`, `tick()`, `move()`, or something similar, although they can often be called anything the programmer would like. Generally what function gets called, and how often it gets called, gets set up in the program's initial entry function (explained in the “What it is” section above).

Includes

WHAT IT IS:

Technically, “includes” takes the contents of another file and dumps them word for word into the file that features the include statement.

Practically, it determines which pre-made functionality you have access to using. It works this way because most of the files we include are ones that define pre-made functions for us to call in our programs – the include statement automatically injects those

definitions to the top of our file. Programming languages come with many libraries prepared for things like displaying text, reading keyboard input, accessing network hardware, and performing comparisons and manipulations of text.

HOW IT LOOKS:

```
#include <stdio.h> // C language for text output
// "stdio" stands for "STanDard Input Output"
```

...then, later in the code...

```
printf("This will show up, thanks to stdio.h!");
```

EXAMPLE OF A USE IN VIDEOGAMES:

Additional libraries can be downloaded from the internet (like Allegro, SDL, or DirectX in this case) that enable you to make simple function calls to load image or audio files into memory, change the screen resolution, display images to the screen, play sound effects, and handle fluid keyboard, mouse, or joystick input. People have earned their PhDs from finding faster ways to get a computer to draw a line – don't waste your time trying to re-invent the wheel, when generations of computer researchers have been

tackling very hard problems so that you won't have to. Instead, find and include libraries that have their brilliant solutions inside them! This allows you to focus your programming on what your videogame does differently with those graphics, sounds, and input events, instead of how to simply get them to work.

Variable Declaration

WHAT IT IS:

A Variable Declaration is a line of code that sets aside a named container for a value. It's up to you as the programmer to come up with a name that you think is suitable. That value is often a number, a letter, a series of letters ("string"), or an object (group of numbers, strings, and/or other objects) – to be saved, updated, and checked within code.

HOW IT LOOKS:

```
int health; // to store/change/check one number
int lives; // store/change/check another number
// int stand for INTeger (any whole number)
```

EXAMPLE OF A USE IN VIDEOGAMES:

Every distinct horizontal position – how far a player, bad guy,

powerup, or particle is from the left side of the screen – must be tracked with a distinct variable. This is true for every distinct vertical position, too.

The same goes for how much health each character with health has, how many characters are alive at once, how much magic power the player has left, which powerups the player has in inventory, what level the player is on... If it's something that has to be remembered even temporarily, and may not always be the same number, it needs a variable. You'll be declaring lots and lots of variables.

Common Types

WHAT IT IS:

A “type” is any form of variable. Some of the most common are `int` (integer, non-decimal number), `float` (floating point, refers to a number with accuracy finer than whole numbers), and `char` (character, i.e. letter).

HOW IT LOOKS:

```
int thisIsAWholeNumber;
```

```
float thisCanBeADecimalNumber;  
char someonesFirstInitialCanBeSavedHere;
```

EXAMPLE OF A USE IN VIDEOGAMES:

The number of lives would be an integer, since it is counted in discrete units. A rocket projectile's speed and position would be floats, since for smooth motion with acceleration finer variation is needed than whole numbers allow. A player's initials on a high score screen would be stored as `char`.

Assignment

WHAT IT IS:

The equal sign in programming, unlike the equal sign in math classes, does not mean both sides of an equation are the same. In fact, it isn't used to write an equation – it's used to create a statement of assignment. Where there's a single equal sign, whatever is on the right side gets evaluated (added together, multiplied out, substituted in, etc.) then stored into the variable to the left of the equal sign. That is, a new value is assigned to that variable name.

HOW IT LOOKS:

```
int temporaryNumberStorage; // creates a variable
temporaryNumberStorage = 5; // sets variable to 5
// read as, "store 5 into temporaryNumberStorage"
```

Here's an example of what NOT to do:

```
5 = temporaryNumberStorage; // meaningless! BAD!
// Say, "store temporaryNumberStorage into 5"
// That sounds like nonsense, right?
// What does it mean to save anything into 5?
// Your computer doesn't know, either
```

And another, slightly fancier example:

```
int num1;
int num2;

num1 = 9;
num2 = 35;
num1 = num1 + num2 - 2;
// num1 winds up 42 (from 9 + 35 - 2)
// num2 stays 35, because it's right of the =
```

EXAMPLE OF A USE IN VIDEOGAMES:

Setting a player's number of lives (assuming "int lives" has been declared earlier in the code):

```
lives = 7;
```

Subtracting 1 life:

```
lives = lives - 1;
```

Giving 2 extra lives:

```
lives = lives + 2;
```

Beware that the following line does not actually change the value of lives, since there is no equal sign acting as an assignment operator (=):

```
lives + 1; // Does not change! No = sign!
```

Since that previous method of adding or subtracting involves typing the variable name twice, there's a slightly faster way that programmers invented to mark that addition or subtraction that saves into a variable with the assignment operator (=):

Subtracting 1 life:

```
lives -= 1;
```

Giving 2 extra lives:

```
lives += 2;
```

You can write it either way. "Var = Var + num;" is the same as typing in "Var += num;" and that also works for multiplication (*), division (/), and subtraction (-). Lastly, the addition or subtraction by 1 is so common in code that there's a way to do that without using the assignment operator:

Subtracting 1 life:

```
lives--;
```

Add 1 life:

```
lives++;
```


(That's where the programming language C++ got it's name. It's the same as the C programming language, but with stuff added on top.)

Object

WHAT IT IS:

A way to organize multiple variables together into a meaningfully named grouping. As with function names and variable names, the programmer invents these names, ideally choosing a name that reflects how the group of variables is intended to be used. In C, it shows up as a struct, which only contains a grouping of variables, but in C++ and most other more recent languages it can be class, which has the ability to also group functions, plus a few other benefits.

HOW IT LOOKS:

```
struct character {  
    int health;  
    int x, y;  
};  
  
character thePlayer;  
character badGuy;  
  
thePlayer.x // storing the player's x position  
badGuy.health // storing a badguy's health value
```

EXAMPLE OF A USE IN VIDEOGAMES:

Objects – whether structs in C or classes in newer programming languages – are essential to keeping organized a project grows to include many different kinds of characters, environment pieces, data formats, and other conceptual groupings of variables. The example above is a very simple, fairly typical use, but could be expanded to account for each character's image, mood, momentum, rotation, animation frame, inventory, etc.

The Array

WHAT IT IS:

An array is a way to store a sequence of variables, such that they share a common label (think street name) but different index numbers to distinguish them from one another (think house or building address).

HOW IT LOOKS:

```
// The number in [brackets] declares array size.  
// Array addresses start at 0, so 1 less than the  
// size of the array is the highest index  
// Defining size [3] gives us [0],[1], and [2]  
  
int numbers[3]; // creates 3 int variable array  
numbers[0] = 1000; // saving 1000 into address 0  
numbers[2] = 70; // it's 3 separate numbers!  
numbers[1] = numbers[2] + numbers[0];
```



```
// numbers[0] winds up as 1000
// numbers[1] winds up as 1070
// numbers[2] winds up as 70

// 3 separate numbers, 1 common label
```

EXAMPLE OF A USE IN VIDEOGAMES:

Any time there are many of something, and each something behaves the same way or based on similar rules, there is probably an array or similar data structure involved. Particle effects, enemy characters, units on the battlefield, projectiles, and so on are often tracked using arrays, utilizing The Object structure of a language to group each individual thing's variables together.

An array of letters ("characters" or chars) forms a "string," word, or phrase which can be used as a name, displayed on screen as text, etc.

If Statement (Conditional)

WHAT IT IS:

Checking whether one or more mathematical comparisons or functions evaluate to true, and only doing a section of code if it is. Valid comparisons include:

$a < b$ (a is less than b?)

$a > b$ (a is greater than b?)

$a \leq b$ (a is less than or equal to b?)

$a \geq b$ (a is greater than or equal to b?)

$a \neq b$ (a is not equal to b?)

$a == b$ (a is equal to b?)

Note that multiple statements can be combined using logic operators:

$(a < b) \ \&\& \ (b > c)$

means (a < b) AND ALSO (b > c)

$(a < b) \ || \ (b > c)$

means (EITHER a < b OR b > c)

HOW IT LOOKS:

```
if(lives > 0) { // "if the player is alive"
    // ...move and draw the player here...
} // otherwise code skips ahead to here
```

EXAMPLE OF A USE IN VIDEOGAMES:

Killing a character when it runs out of health, forcing the player to stay on screen if the x or y positions go outside an acceptable range, checking whether a gun needs to be reloaded before allowing it to initiate the reload animation/code, etc.

Else and Else-If Statements

WHAT IT IS:

Using a section of code only if the result of an if statement turned out false.

HOW IT LOOKS:

```
if(lives > 0) { // "is the player alive?"
    // ...move and draw the player here...
} else if(continues > 0) { // "continues?"
    // ...go to the continue screen...
} else { // "if the player isn't"
    // ...play sad music...
    // ...go to game over screen...
}
```

Note that you can have as many else-ifs as needed. You can even just have an else first thing right after an if. The presence of any else or else-if is optional. If there's an else, it must come last in the chain.

EXAMPLE OF A USE IN VIDEOGAMES:

It's extremely common to use this type of series of evaluations in games for a computer controlled enemy to evaluate circumstances with a pre-determined priority, to handle a series of circumstances that might cause a collision to detonate a projectile, or to handle menu item selection.

The switch-case sequence (not covered here, but easy to find on

the internet) is an alternate construction available if the else-if chain is checking matches against consecutive integers (is it 1? otherwise, 2? otherwise, 3? etc.).

While Loop

WHAT IT IS:

It works a lot like an If Statement, except that when its closing brace is reached by the code, it checks the While condition again. When the While condition is still true, it goes through the code between its braces another time; if the while condition's comparison expression is false, then code continues after its closing brace.

HOW IT LOOKS:

```
while( gameHasBeenLost == 0 ) {
    useMouseInput();
    moveEnemies();
    updateScreen();
    if( health < 0 ) {
        gameHasBeenLost = 1;
    }
}
```

EXAMPLE OF A USE IN VIDEOGAMES:

Its most common usage is shown immediately above – the game's logic and screen updates occur constantly within a while loop, until something happens within the program (a key is pressed, a

menu item is clicked, lives run out, etc,) causing its comparison expression (here “gameHasBeenLost IS EQUAL TO 0”) to no longer be true.

Take heed that with any loop, the risk of an “infinite loop” – code that never stops executing – appears if nothing inside it happens that will turn its expression condition false. If your program is being run in an old terminal, and seems to be stuck, CTRL+C will bail from it. If all else fails, CTRL+ALT+DEL (PC Windows) or Force Quit (Mac) will easily and safely terminate a program that is trapped in an infinite loop.

Note that the above usage is no longer common in certain newer languages, particular ActionScript 3, which instead handles game logic best in the form of setting up a timer to call a particular function some number of times per second. (Not shown here.)

For Loop

WHAT IT IS:

The For Loop groups together a common and very useful pattern of programming: set up a variable to be used in a while loop, declare the comparison expression for the while loop to evaluate, and do something each time through the loop to affect the loop’s variable. It’s commonly demonstrated as an easy way to count from one number to another, but it can also be used to scan across every letter in a word, check every pixel of an image, or call a function on every bad guy/item in the world.

HOW IT LOOKS:

```
for(int count=0; count<100; count++) {  
    // ...do something here...  
    // it will happen 100 times.  
}
```

...that behaves exactly the same as this:

```
int count=0;  
while(count<100) {  
    // ...do something here...  
    // it will happen 100 times.  
    count = count + 1; //Same as "count++;"  
}
```

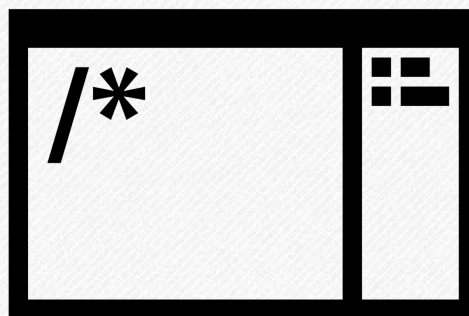
EXAMPLE OF A USE IN VIDEOGAMES:

Going through each bad guy and updating their position, where

each bad guy is an object in an array, by calling a function on the bad guy in the array position denoted by the for loop's counting variable.

HOW PROGRAMMERS PROGRAM

This is something rarely taught, because it's mostly learned through experience and hard to show in books or example code. But I think it's important to emphasize that code is rarely written top-to-bottom in one pass.



NOT FROM START TO FINISH

As a beginning programmer, looking at finished example code can seem daunting. How, with any amount of experience, does someone know what key to type next? What is the thought process in someone's head while they're typing out new functions, classes, and algorithms?

Unless a programmer is retyping something that has been committed to memory, the characters aren't typed in their final order. Finished code for

anything other than the most trivial of examples typically involves dozens, sometimes hundreds of iterations, each of which amount to drafting, editing, and polishing.

A Steven King novel is an impressive thing to read. A Spielberg film is an impressive thing to watch. Every piece seems well planned, and every scene throws something new our way. The pure density of ideas presented can seem overwhelming to an aspiring creative. But that quality of

content doesn't just come together that way the first time, beginning to end. The final product came together only after countless drafts, aggressive editing, and years of attention paid to polishing every last detail.

Why would computer programming be different?

FROM THE OUTSIDE IN

The first time that I compile any videogame project, the first version is a black screen. The second version that I compile is a black screen that makes a sound, or displays a rectangle. The third version will perhaps make the sound when spacebar is pressed, or move the rectangle to the last place the mouse clicks. Until those things work right, the videogame project isn't going to work anyway.

But that isn't just how the process starts – that's also how features get implemented. Whether planning went into which features are worth making (a good habit if you have enough experience to

estimate how long different features will take, or how valuable they'll be in the big picture), or whether it's just time for hacking features together to explore and entertain creative whims (a good way to learn), the actual process is still incremental. A version is built with an unanimated character drawn on screen, the next version moves that character according to which control keys are pressed, the next version applies gravity to the character, followed by a jump ability, then collision with enemies, followed by hook up to animations, and so on. Between each step, there's compilation and brief playtesting to make certain that the last executed layer does what it's expected to do.

ENABLE FAST ITERATIONS TO TEST CHANGES

Because of the iterative nature of this process, consideration should go into how to streamline the testing cycles early on in development. While the tendency of many beginners is to first make the logos, splash screen, and

menus – again demonstrating the erroneous thinking that something ought to be made in the order it will be experienced in – I suggest working in the opposite order.

Leave the interfaces and splash screens for last, when they'll slow down testing the least possible cycles, letting the game compile or run directly into play mode as early as possible and keeping it that way for the bulk of the development time.

ERRORS ARE A SIGN OF PROGRESS, NOT FAILURE

Another part of the process in how programmers program is hitting frequent errors and obstacles.

Whether we're setting up the development environment, writing core functionality, or extending existing functionality to tend to details, the process of programming is often a two-way dialogue with the machine.

We spend years in schools learning to associate red marks or answers marked wrong (especially to be redone!) as a sign that we

have done something wrong. This thinking has no place in programming, particularly when the notification of what's "wrong" just comes from the machine.

YOUR PATIENT TEACHER

The compiler can be thought of as a teacher with infinite patience.

The compiler is always happy to proof-read code, and after passing that grammar/spelling inspection, it will show exactly what the work so far does.

It's a bad idea to wait until most of an work is done before making sure that it's being done right. If starting in the wrong direction, it'd be wise to find that out as soon as possible before the mistake grows throughout the program. When it comes to asking real human beings to look over what we've done – whether we're talking about parents, peers, teachers, or online forums – we have a carefully conditioned feeling that we should pester them only if we have to, because we have to

respect their time and not interfere with their obligations.

The compiler doesn't have any plans for later.

MATCHING {BRACES} WHILE TYPING

Much of the math that we do “on paper” is a process of jotting as much as we can on the page so we can devote our mental energy to thinking through the steps of (a.) what else to jot on the page and (b.) what are the proper processes to advance what's on the page to the next step. Even when we do mental math, most of us still wiggle our fingers to temporarily hang on to information (like carry digits) while we juggle other figures and steps in our head.

That's how the most expert programmers program. It's called “distributed cognition” by psychologists, so named because it distributes cognitive processes to include mechanisms outside the mind – but remembering the fancy word for it is less important than practicing it.

When typing a loop, function, or conditional, most programmers close braces pairs and quickly handle format indenting before the filling what goes on the inside. In other words, when I'm halfway through typing this:

```
for(int i=0;i<100;i++) {  
    printf("The secret password is XYZZY");  
}
```

It doesn't look like this:

```
for(int i=0;i<100;i++) {  
    printf("The secre
```

It looks like this:

```
for(int i=0;i<100;i++) {  
}
```

The distinction looks subtle, but its importance and difference becomes more clear when nestling two loops to perform a computation over all entries in a two dimension array, and checking with a conditional inside them, all within a function. Still, each matching right-brace "}" is typed immediately after the corresponding left-brace "{", so that the code is typed in the following order from 1st to 11th:


```

int some2DArray[100][100]; // from early in
code...
int some2DArray[100][100]; // elsewhere in code
void ValidateTheArray() { // 1st line typed
    for(int y=0;y<100;y++) { // 3rd line typed
        for(int x=0;x<100;x++) { // 5th line
            if(some2DArray[y][x] == -1) { // 7th
                // TO-DO NEXT: type 10th line(s)
            } else { // 8th line typed
                // TO-DO LAST:type 11th line(s)
            } // 9th line typed
        } // 6th line typed. Matches 5th line
    } // 4th line typed. Closes braces from 3rd
} // 2nd line typed, Right half of 1st line typed

```

Closing out the braces and doing indentions before filling inside them in this case means fewer things to remember while thinking through what needs to be done inside the nestled loops. We thought about that part, put it down in code (like jotting numbers down for long division), enabling us to forget about it while we focus on writing the code at the innermost scope.

This is a particularly powerful habit because a mistake in matching braces can be one of the most difficult for the programmer, or even the compiler, to accurately diagnose. Because a mismatched brace changes what parts of the

code relate to one another, the compiler may think that the error is someplace else far away in the code, or even give the unhelpful "expecting right brace at end of program" which means there could be a "}" missing anywhere in the code. Time not spent getting frustrated tracking down these hard-to-fix but easy-to-avoid code errors is time that can instead be spent productively iterating on functionality.

SCAFFOLDING: TEMPORARY WORK IMPROVING LASTING WORK

The last part we'll cover here on how programmers program is to take advantage of the concept of scaffolding. Scaffolding in everyday use, for anyone less familiar with the term, refers to the temporary layers of ladders and wood-planks stacked in metal frames around the outside of a building while it's under construction, giving workers easy reach to everything. Once the building is completed, the scaffolding is torn down and removed, leaving behind a

structure which could not have been built as efficiently or with as much attention to detail had scaffolding not been used.

What I call scaffolding, in the world of programming, is what most developers call "debug output". The idea is to temporarily include extra lines of code to help make visible the otherwise invisible calculations and flow of the program. Print statements (`printf` in C/C++), trace statements (ActionScript 3), log statements, and the like outputting information to the console like, "starting function GameDraw", "loading background image Swirl03", or "left mouse click detected" are examples of this.

DEBUGGING

A good debugger and debugging environment, when available, provide this same functionality through breakpoints, although depending on the language, environment, and platform, a good debugger may not always be available. Breakpoints pause the

program at a given instruction, enabling the program to advance one step at a time, observing numbers and changes in memory. This method, when available, is particularly well-suited to making sense of obscure technical bugs.

Breakpoints, like debug print statements or visualization of internal states, are removed or disabled from the game or source code before shared. Debug elements are not needed by the time a game or source code can be shared, because like the scaffolding around the new building, they have already been put to use in ensuring that the project itself came together.

Visualizing this information in-game can also be helpful sometimes when rooting out questions about real-time functionality that takes place across multiple frames which could be difficult to read from lines of text, such as drawing the player as a different color rectangle based on whether the collision

code currently thinks that the player is on the ground.

INTERNAL TOOLS

The greatest example of the scaffolding concept is the development of software that simplifies the development of software - level design tools, weapon/unit stat editing tools, animation tools - programs that enable design problems to be solved using design tools instead of through programming. Though they're (usually) not shipped with the final game, this additional programming work makes it possible for the game to be finished sooner and better. In smaller games - including most of my old 2D tile-based projects - sometimes the easiest way to build the level editor is to introduce it as a level editing mode in the game itself, hiding that functionality for the final release. That way the same assets (graphics/sounds), rendering code, and level formats are the same for

both playing and editing as the code base develops.

When looking at example source, or playing a finished game, rather than thinking about how the entire program could have come together at once (a daunting task no matter how much experience someone has!), be thinking instead about how scaffolding of debug statements could verify aspects of the program while working on it, or about what the first few crude features would be developed and tested after the empty screen compiles. Test, then repeat.

START FOCUSED ON WHAT'S INTERESTING

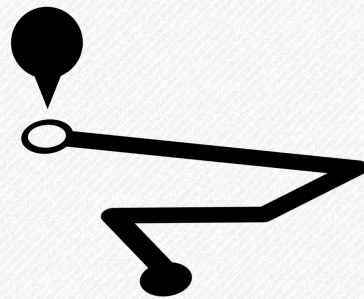
Think about music composition - the refrain or underlying tune is likely to be worked on before the beginning or end. Or a story might be built around one or more intense climactic scenes, which will need to be packaged between build up and resolution before being ready for others.

When an idea jumps into mind for a videogame, starting to write what the final videogame code

might be like from start to finish - complete with rigorous class definitions for everything and all the trappings of a finished game's functionality - just doesn't make sense. Get the bare core working with as little code as possible at first, then layer and iterate.

POSITION AND SPEED VARIABLES

Easily the most common uses of variables for game programming is in simply storing and updating object positions or movement directions. There's nothing magic in how this works. Let's break it down here.



The impression of spatial continuity – that an object exists at some location, and stays there until it moves to an adjacent position – is not assumed by a computer. This seemingly natural behavior has to be described in code by the programmer.

Fortunately, there is a pattern for this.

It can seem obtuse the first time or two it's seen – it did for me – though once used a few times its workings become clear.

OVERVIEW

What follows assumes very little prior experience, so there's plenty of text in an effort to be thorough. There are only a few points, though:

1. Create number variables to store the position of the thing to be drawn.
2. Use those variables as coordinates when drawing the image for that thing.
3. Change those variables to move it.

4. Create additional number variables – to treat as speed for horizontal and vertical movement – and add those to the position variables every frame.

DRAWING AT COORDINATES

Independent of the programming language and library, there's a function to draw a specified shape or image to the screen at some coordinate.

As mentioned in Chapter 1, Section 6 (General Concepts for Beginning Developers), the screen is a 2D plane, often measured in pixels as distance horizontally and vertically from the top-left corner.

Typical draw functions range from plotting a single pixel, to drawing a circle or rectangle, to copying a loaded image file to the screen.

For example purposes, we'll refer to the image drawing function in this form:

```
copyBitmapToPosition(10,15,goblin);
```

...to signify drawing the image pointed to by the variable named “goblin” with its top-left corner 10

pixels from the left side of the screen, and 15 pixels from the top of the screen. To be clear, that's a fictional, pseudo-code notation – actual function name and order of values vary by programming language and graphics library.

LOADING THE IMAGE

Before that example line could be used in code to draw a goblin, we would need to declare that there's an image we're referring to in code by the label “goblin”, as well as specifying which filename to load into that label. Example code:

```
Image goblin; // creates 'goblin' image label
```

...skipping unrelated code...

```
goblin = load_png("monster.png"); // loads image
```

(Remember: any writing after the // marks implies comments, meaning that text is ignored by the compiler when code is translated into an application.)

CLEARING EVERY FRAME

Computer games update the screen dozens of times every second.

The first thing to do with every update is to clear the screen, giving backdrop to the object(s) and concealing the previous draw. The two easiest ways to do this (either works, no need to do both):

1. Draw a filled rectangle, at the top-left corner of the screen, the same dimensions as the screen. (“background color”)

2. Draw an image, one with the same dimensions as the screen, lining up its top-left corner with the top-left of the screen. (“background image”)

An example of the latter, using our draw function, might look like this:

```
copyBitmapToPosition(0,0,sky);
```

That erases whatever the world looked like the previous frame – which is why we do it – but that also means we need to constantly redraw the image at its current position for it to show.

DRAWING EVERY FRAME

After the part that clears and resets the screen, we call the function that draws the image:

```
copyBitmapToPosition(10,15,goblin);
```

In Processing, that means we’d put the copyBitmapToPosition call in the draw() function, which gets called every frame. In C++ or AS3, the call would go in whichever function we’ve attached to a timer to be called 24-60 times per second, somewhere after the line that draws the background to clear the screen.

KEEPING TRACK OF POSITION

The problem with a call like this one:

```
copyBitmapToPosition(10,15,goblin);
```

...is that it always draws the goblin image at the same position on the screen. This is fine for drawing the background every frame to clear the image, but objects like goblins should be able to move.

To support that movement, we’ll create two number variables to store the goblin’s position along each axis:

```
int gob_x = 10, gob_y = 15; // only once in code
```

...skipping middle code...

```
// called every frame to draw image at position  
copyBitmapToPosition(gob_x, gob_y, goblin);
```


Note that the `gob_x` and `gob_y` variables are now used in the `copyBitmapToPosition` as the position to draw the image, rather than specific numbers. This way, when `gob_x` is changed, whether dramatically (`gob_x = 450`) or incrementally (`gob_x = gob_x + 2`), the position where `gob_x` is being drawn for future frames will be changed.

Changing the value in `gob_y` will likewise affect the vertical spot where the goblin image is being drawn.

Small changes – say, adding or subtracting a few every frame – creates the impression of movement. Larger changes, mid-gameplay, suggest teleportation, although setting values at the start of a level is unnoticeable without reference point, and the way to set initial positions.

CONSTANT MOVEMENT

Just like the problem with this:

```
copyBitmapToPosition(10,15,goblin);
```

...was that the position was always the same (“hard coded”), the problem with this:

```
int gob_x=10, gob_y=15;
```

...later in the code...

```
gob_x = gob_x + 1; // move right by 1 pixel
copyBitmapToPosition(gob_x,gob_y,goblin);
```

...is that the speed is always the same (1 to the right every frame). Instead, we add another pair of variables, denoting the current horizontal and vertical speed:

```
int gob_x=10, gob_y=15;
int gob_x_moveAmount=0, gob_y_moveAmount=0;
```

...and later in the code...

```
if( keyPressed( RIGHT_ARROW) ) {
    gob_x_moveAmount = 1; // right movement
} else if( keyPressed( LEFT_ARROW) ) {
    gob_x_moveAmount = -1; // left movement
} else { // neither left nor right arrows pressed
    gob_x_moveAmount = 0; // stop horizontal move
}
```

```
if( keyPressed( UP_ARROW) ) {
    gob_y_moveAmount = -1; // upward movement
} else if( keyPressed( DOWN_ARROW) ) {
    gob_y_moveAmount = 1; // downward movement
} else { // neither up nor down arrows pressed
    gob_y_moveAmount = 0; // stop vertical move
}
```

```
// each frame, adjust position by current speed
gob_x = gob_x + gob_x_moveAmount;
gob_y = gob_y + gob_y_moveAmount;
```

```
copyBitmapToPosition(gob_x,gob_y,goblin);
```

By keeping horizontal and vertical movement stored separate from position, we can program and reason differently about movement vs position. For example we can

now zero – or reverse – speed when the goblin bumps into something. We can adjust speed when the goblin stands on a conveyor belt, adjust speed based on wind vectors, fling the goblin backward from an explosive blast, etc. by modifying the variables for speed.

SMOOTHER MOVEMENT

In the example above, even though the code is just pseudo-code (not actual wording for any particular language, just showing the pattern), the common ‘int’/integer type was used for position and speed. Although pixel positions are whole numbers (501, 502, 503...), decimal real values of the float/double/number variety work better for keeping track of position and speed values.

When the speed values are decimal, percentage falloff can be used to slowly decay speed:

```
float gob_x = 0.0;
float gob_x_moveAmount = 0.0;
```

...skip position variables and key code...

```
// move by whatever speed amount remains
gob_x += gob_x_moveAmount
```

```
// decay speed by 7% every frame:
gob_x_moveAmount = gob_x_moveAmount * 0.93;
```

With code like this, rather than instantly coming to a halt when no keys are pressed, the object will glide gracefully to a stop.

CODE ORGANIZATION

Classes can be used to better organize the data above. X/Y pairs are often contained in a Point or Vector class. Both vectors are often grouped into a larger class, along with the image used (goblin, in this case), plus the functions used to handle the character’s movement and updates on the screen.

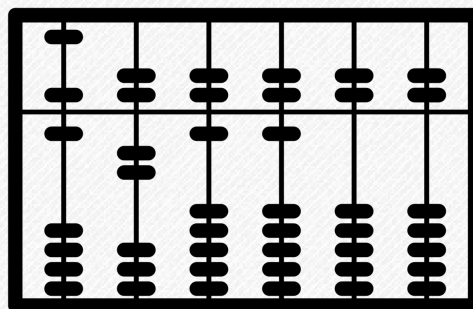
That amount of structure isn’t necessary to get it working, but it’s useful to stay organized and minimize mental overhead while expanding the program to include multiple moving objects. If you’re new to programming, consider this a note which will make more sense in time.

OTHER USES OF POSITION VARIABLES

Keeping the position stored in variables is also handy for comparing the character's location to other values – screen edges, world edges, and most often, the position values used to track other objects. When the distance between the coordinates of one character and another is within some range, a function for handling collision between those characters can be called.

FLOAT AND INT VARIABLES: CASTING AND OTHER ISSUES

Most coding environments distinguish numbers that can't have decimal parts from those that do. Why don't we always want the extra precision? Here I spell out how and when one or the other option works best.



To programmers with experience making games, this may seem like something that doesn't need to be covered. As I've pointed out before though, that's a sign of a common barrier to people getting started: it's a detail which is quite unnatural, however because it seems obvious to people that are used to it, it is rarely brought up.

One of the first things learned in programming is that there are different variable types for numbers with decimal precision (typically of type float, double, or

number) versus numbers without decimal precision (almost always using the type int, short for integer).

For readers still getting used to the distinction, numbers with decimal precision include:

```
0.0
-1.5
3.14159265
9000.36
57.3333333
```

Meanwhile, examples of int values include:

```
0
-2
3
9000
57
```

Note that a number with decimal precision can have a decimal part of .00000... but it's of a different type because it could have a decimal part assigned, not because it currently does. As a departure from pure math terminology, simply setting a float value to 6.0 does not change that float value to an integer in the programming sense.

For this reason, in programming putting a “.0” after a number communicates to the compiler (and other programmers) that a number is intended as a float/number with decimal precision, not as type int. I use that notation throughout this entry. 300 is different from 300.0, because 300 is of type int whereas 300.0 is of type float.

DON'T WORRY ABOUT PERFORMANCE (YET)

While getting started out, and well into making a number of games, I'd suggest not worrying about performance when deciding what types of variables to use.

Decades ago, before floating point units were common in processors, using int for nearly everything was necessary for performance.

Computer science trickery called fixed-point math enabled programmers to simulate decimal accuracy, but at the expense of reduced precision and range. That technique is still sometimes needed for making games for cellphones of the non-smartphone variety.

Anymore, calculations with floats are nearly the same speed as integer math, and in either case it's done fast enough to not be a primary performance concern. The minor performance differences now come down to cache complexities and frequency of casting between types, both of which are dramatically overshadowed by other issues.

Especially with the number of objects being juggled for a typical hobbyist computer game (dozens of moving things, rather than thousands), basic variable choice

is unlikely to be a source of problems.

AVOID CHECKING EQUALITY ON FLOATS

Float values do not have infinitely fine precision. One effect of this is that a series of math operations which, done on paper, should equal exactly 5.0, may on a computer yield a number extremely close to, but not, that value, ex. 4.999999999 or 5.000000001. Rather than checking:

```
if(someDecimalVariable == 160.0)
```

...which risks being false due to small but accumulating precision errors, something along the lines of this is safer:

```
if( absValue(someDecimalVariable-160.0) <= 0.1)
```

...since it allows for a range of minor error. I am using `absValue(number)` as a generic math function for absolute value, though depending on language it's often of the form `abs(number)` or `Math.abs(number)`.

Note that if context allows, inequalities are much safer than

exact equality, so it's perfectly fine to check:

```
if(someDecimalVariable >= 160.0)
```

...or...

```
if(someDecimalVariable <= 160.0)
```

SUITABILITY OF EACH TYPE

If it can be counted in definite pieces, use an int. This typically applies to ammo, score, or the number of anything (enemies, particles, etc.). Indexes into an array always need to be done by integer - for an array with 8 values in it, it makes sense to access `someArray[4]` or `someArray[5]`, but there's no value saved at index 4.3, meaning `someArray[4.3]`.

Health can be either, depending on how the game works. If the healthbar is presented as a percentage, as is often the case of a 1-on-1 fighting game, decimal precision is an option. If health is discrete units - for example each tank has 5 boxes of health (even if perhaps a big explosion might take away 2 or 3 at once), saving the health as an int makes more sense.

Positions and speeds are the most common values to be stored and manipulated with decimal precision. A player's character might have position (10.2, 200.0) - whether as the .x and .y on a Point/Vector object or as two separate variables like positionX and positionY - and velocity (3.25, 0.0) - likewise either on a Point/Vector object or split into velX and velY. Motion would be simulated by adding the velocity to the position each frame, so that in the next frame the player will be at (13.55, 200.0), creating the illusion of movement. Keyboard or gamepad controls, drag calculations, and other effects on player movement would then modify the velocity rather than affecting position directly.

PROBLEMS WITH INT MOVEMENT

Using decimal precision for positions and velocity can seem counter-intuitive at first. This is especially the case for position, since positions on screen in a 2D game are typically distinguished

by pixel coordinates. A player's character might be drawn at position (15, 200) one frame, then as they move right slightly, the character would be drawn instead at (16, 200). There is no such thing as a pixel 15.3 from the left edge.

[As a brief aside: using textured polygons and a generalized rendering approach, it's certainly possible to draw something at position (15.3, 200.0) through a bit of pixel blurring/sampling, but this is being written regarding the typical beginning case of 2D games in which bitmap data gets copied to directly to video memory.]

The first time someone integrates keyboard controls into a project made from scratch, it typically begins by figuring out how to draw a graphic to the screen at a given position:

```
// this is pseudocode, not real function names!  
// blits graphicToDraw to position  
void draw(bitmap graphicToDraw, int drawAtX,  
          int drawAtY);  
  
// ...skipping over code, into main loop...  
while(gameInPlay) {  
    draw(playerGraphic, 125, 300);  
}
```



```

// update screen from buffer
// (not relevant to all languages)
updateScreen();
}

/* reminder that for many languages, instead of
a while(gameInPlay) loop, this type of logic
takes place inside an update() function called
by timer event ~30 times per second */

```

Realizing that a moving player character will need its position saved within variables, rather than hard coded, the code will then change into something like this:

```

int playerX = 125;
int playerY = 300;

// ...skipping over code, into main loop...
while(gameInPlay) {
    draw(playerGraphic, playerX, playerY);

    updateScreen();
}

```

This way, the position of the player's graphic can be changed by incrementing or setting the value of playerX and/or playerY, like so:

```

int playerX = 125;
int playerY = 300;

while(gameInPlay) {
    if(keyboard.leftArrowPressed) {
        playerX--;
        // above is short for playerX = playerX -
1;
    }
    if(keyboard.rightArrowPressed) {
        playerX++;
    }

    if(keyboard.upArrowPressed) {
        // recall that (0,0) is top-left
        // smaller Y in screen coords means
higher
        playerY--;
    }
    if(keyboard.downArrowPressed) {
        playerY++;
    }
}

```

```

}

draw(playerGraphic, playerX, playerY);
updateScreen();
}

```

Based on the code above, the player character can be shuffled around the screen using arrow keys. This works, technically, but it's very jerky. There are three major shortcomings:

- It's impossible to give the player a speed other than full pixel increments per cycle, i.e. its minimum speed is 1 pixel/frame.
- It's impossible in this implementation to slow down gradually to a stop.
- It's impossible to move the player at an arbitrary angle, for example 30 degrees or -114 degrees, while maintaining the same speed.

The solution to those problems is to use decimal precision (float or number) types instead of int for position.

```

float playerX = 125.0;
float playerY = 300.0;
float playerSpeed = 1.0;

while(gameInPlay) {
    if(keyboard.leftArrowPressed) {
        playerX -= playerSpeed;
    }
}

```



```

        // short for playerX = playerX - player-
Speed;
    }
    if(keyboard.rightArrowPressed) {
        playerX += playerSpeed;
    }

    if(keyboard.upArrowPressed) {
        playerY -= playerSpeed;
    }
    if(keyboard.downArrowPressed) {
        playerY += playerSpeed;
    }

    draw(playerGraphic, playerX, playerY);
    updateScreen();
}

```

Now, we could set `playerSpeed` to 0.4 to move at only 40% the speed of 1 pixel/frame. We can increase the value of `playerSpeed` during the game ("playerSpeed += 0.2;" when we want speed to increase by 0.2 pixels per frame) to make the player move faster, and decay it gradually to make the player smoothly lose speed (using a line like "playerSpeed *= 0.98;" to lose 2% of our speed every frame). Since the trig operations `sin()` and `cos()` return values of decimal precision, we can now adjust `playerX` and `playerY` by those if we'd like to move the player by some arbitrary angle.

Now there's just one problem: the compiler is probably griping at us with warnings about implicit

casting between types, or even throwing an error for failing to manually between cast types.

CASTING BETWEEN TYPES

Casting is the operation of converting a value in memory from one type to another. For our purposes here, it means taking an int value like 5 and converting it to the float value 5.0, or in the other direction, taking a float value of 5.3 and converting it to an int value of 5.

Syntax for casting varies by programming language, and many programming languages support multiple ways to cast variables, but it often looks something like this:

```

float someDecimal = 5.6; // setting a value
int ourCastedValue = ((int)someDecimal); // 5
// note: cast drops decimal, it doesn't round!

// or going the other direction:

int someInt = 6; // setting up a value to cast
float otherCastedValue = ((float)someInt);
// otherCastedValue will be 6.000000...

```

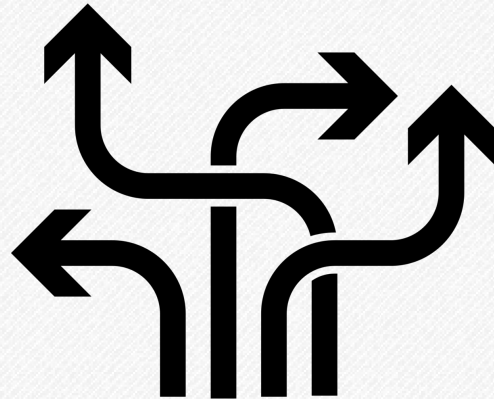
The cast does not need to be done when assigning a value, it can also be done in-line, so that we could update our draw code from earlier to look like this:

```
draw(playerGraphic, ((int)playerX),  
                    ((int)playerY) );
```

Doing it this way would remove the compiler's warning or error, since we're now specifically acknowledging that we deliberately mean to use a decimal-precision number as an int value here, i.e. that we are okay with losing the decimal portion of the value for this purpose.

HACK THEN REFACTOR

One of the disadvantages that new game developers sometimes struggle with is recognizing that the “best” way to do it is sometimes not what’s fastest for the program, but instead what’s fastest for the programmer.



ROUGH DRAFTS SHOULD LOOK ROUGH

One of the common challenges holding up beginning game developers, much like it holds up beginning writers, is trying to get everything down right the first time, perfect immediately, as soon as it’s “on the page.” A rough draft ought to be rough, whether it’s written in code or in words. It’s the best time to be taking some chances, hashing out some incomplete thoughts to see where (if anywhere) they lead, getting it on screen as typed characters to

then figure out in the following steps step how it might be better rearranged, chunked, and reorganized.

Schooling introduces everyone to some basic process steps to apply this process to writing. For practical game coding we’re mostly on our own.

STEP 1: JUST GET IT WORKING ASAP

When just getting started, give up pretense for elegance or doing things the proper way and just embrace the hackiness while figuring out what you want. Even if

this is the first step of a larger projects, treat this step like game jam or timed programming challenge, just get it working and out of your head. Don't worry yet about optimization, what you'll keep, code readability, or organization, just get it working to be able to make a more informed choice about what's worth keeping. During initial prototyping, treat it less like an engineering task and more like brainstorming, sketching, or outlining. My early iterations when getting a game working can tend to look a little like this (though handling all four movement directions instead of just one, and usually for a less trivial interaction):

```
//// just pseudocode, not C++, AS3, Java, etc.!!
int playerX, playerY, enemyX, enemyY;
int enemyAlive = 1;
int playerHealth = 1;
Bitmap backgroundFile; // load elsewhere in code

function everyFrameTick() { // just for example
    if(keyboard.holding(RIGHT)) {
        playerX++;
    }
    if(playerX > screenWidth) {
        playerX = screenWidth;
    }
    if(distance(playerX,playerY,enemyX,enemyY) <
                                   collisionRange) {
        playerHealth--;
        enemyAlive = 0;
    }
    drawImage(backgroundFile);
    if(enemyAlive != 0) {
        drawCircle(enemyX,enemyY,color.RED);
    }
}
```

```
    }
    if(playerHealth > 0) {
        drawCircle(playerX,playerY,color.GREEN);
    }
}
```

STEP 2: LABEL PARTS WITH COMMENTS

That first step can be good for ideation, but typically results in large undifferentiated blocks of code serving mixed purposes. This definitely doesn't scale well to a larger, distributed, or more elaborate program, but while just getting traction it has its advantages. Having all the code in one place like that can make it very quick and easy to make sweeping changes to what you're doing and how, ways that data affect one another, and there's no time spent hopping between files. The code at this point is small enough to keep in mind all at the same time, and using CTRL+F in the text editor to jump to spots works fine but probably isn't even necessary.

Don't beat yourself up over a rough draft looking like a rough draft. That is how it should be. Add short, one-line, functionally descriptive comments breaking up

sections of that code. “Noun verb” form works best for this, such as “// application setup” or “particle effects update.” Rearrange line order to better group them if necessary without changing functionality. One section might be “// player movement” or “//// player bounds checking” – using multiple comment pairs as in that second example to mark subsections just as you would for an outline.

```
int playerX, playerY, enemyX, enemyY;
int enemyAlive = 1;
int playerHealth = 1;
Bitmap backgroundFile; // load elsewhere in code

function everyFrameTick() {
    // wipe screen by redrawing background
    drawImage(backgroundFile);

    // enemy movement and draw
    //// collision check between player and enemy
    if(distance(playerX,playerY,enemyX,enemyY) <
collisionRange) {
        playerHealth--;
        enemyAlive = 0;
    }
    //// draw enemy
    if(enemyAlive != 0) {
        drawCircle(enemyX,enemyY,color.RED);
    }

    // player movement and draw
    //// player keyboard input
    if(keyboard.holding(RIGHT)) {
        playerX++;
    }
    //// player bounds checking
    if(playerX > screenWidth) {
        playerX = screenWidth;
    }
    //// draw player
    if(playerHealth > 0) {
        drawCircle(playerX,playerY,color.GREEN);
    }
}
```

STEP 3: CUT INTO FUNCTIONS AT COMMENTS
Carve that larger body of code into functions named after the comments you made to split up groups of the code. The code going from “// player movement” up to the next “//” top-level depth comment should be moved to a new function called “playerMovement()” which, in the example which had a subsection marked as “//// player bounds checking” could be further split to call a function called “playerBoundsChecking()” and so on. If variables are needed among both, consider pulling them out temporarily as globals (if that wasn’t done already as part of rapid prototyping) with similar names (playerX, playerJumpHoldTimer...).

```
int playerX, playerY,enemyX, enemyY;
int enemyAlive = 1;
int playerHealth = 1;
Bitmap backgroundFile;

function redrawBackground() {
    drawImage(backgroundFile);
}

function enemyMovementAndDraw() {
    enemyPlayerCollisionCheck();
    drawEnemy();
}

function enemyPlayerCollisionCheck() {
    if(distance(playerX,playerY,enemyX,enemyY) <
collisionRange) {
```



```

        playerHealth--;
        enemyAlive = 0;
    }
}

function drawEnemy() {
    if(enemyAlive != 0) {
        drawCircle(enemyX, enemyY, color.RED);
    }
}

function playerMovementAndDraw() {
    playerKeyboardInput();
    playerBoundsChecking();
    playerDraw();
}

function playerKeyboardInput() {
    if(keyboard.holding(RIGHT)) {
        playerX++;
    }
}

function playerBoundsChecking() {
    if(playerX > screenWidth) {
        playerX = screenWidth;
    }
}

function playerDraw() {
    if(playerHealth > 0) {
        drawCircle(playerX, playerY, color.GREEN);
    }
}

function everyFrameTick() {
    redrawBackground();
    enemyMovementAndDraw();
    playerMovementAndDraw();
}

```

STEP 4: GROUP FUNCTIONS INTO FILE/CLASS

For functions and variables that have a common root – “player” in the case above – consider whether that can be better split off into its own class, or at least its own file. Likewise global variables that use a common root can likely be moved into being member variables on the classes that share the same noun.

```

////// FILE: player.??? ////
class Player {

```

```

    int x, y;
    int health = 1;

    function playerMovementAndDraw() {
        keyboardInput();
        boundsChecking();
        drawMe();
    }

    function keyboardInput() {
        if(keyboard.holding(RIGHT)) {
            x++;
        }
    }

    function boundsChecking() {
        if(x > screenWidth) {
            y = screenWidth;
        }
    }

    function drawMe() {
        if(health > 0) {
            drawCircle(x, y, color.GREEN);
        }
    }
}

////// FILE: enemy.??? ////
class Enemy {
    int x, y;
    int alive = 1;

    function movementAndDraw() {
        collisionCheckAgainstPlayer();
        drawMe();
    }
    function drawMe() {
        if(alive != 0) {
            drawCircle(x, y, color.RED);
        }
    }
    function collisionCheckAgainstPlayer() {
        if(distance(player.x, player.y,
                    this.x, this.y) <
                    collisionRange) {
            player.health--;
            alive = 0;
        }
    }
}

////// FILE: main.??? ////
Bitmap backgroundFile; // loading jpg elsewhere
in code
Enemy testEnemy = new Enemy();
Player thePlayer = new Player();

function redrawBackground() {
    drawImage(backgroundFile);
}

function everyFrameTick() { // incomplete example
for illustration!
    redrawBackground();
    testEnemy.MovementAndDraw();
    thePlayer.MovementAndDraw();
}

```

Many programmers have the inclination - or rather the training - to start at this last step instead of the mess that I illustrated at the first step. If you're planning a telecom infrastructure, a compiler, or banking software, by all means there's really no room for even temporary hackiness. However if you're prototyping a design that's rapidly changing early on based on how it feels and plays, pulling together basic functionality just to get a demo on the screen, consider for a moment how much more bloated the last iteration of code on this page looks than the first (61 instead of 25 lines, nearly 150% more lines), despite doing essentially the same thing.

It's probably not immediately clear from this simplified, partial code example just how much trying to start at the more organized phase can slow down iteration early on, but extremely rapid iteration on all aspects of the design (not just tuning values, but adding or rethinking mathematical and

causal relationships with new or different values to tune!) can go a long way when a project is just forming. The extra minutes and mental energy spent on playing with information freely like clay, directly and all in the same place, rather than mentally moving from room to room simply to get things wired together at all quickly adds up.

Hack while figuring out what you're doing, then refactor once you begin to develop a clearer sense for how you want it to feel and work going forward.

TRADEOFF IN THE TRANSITION

After applying this process to break up the code, someone else will be able to read it more easily, but it also renders it a little less malleable for rapid prototype iterations. The added layer of structure can begin to embed relationships and assumptions. This is even more the case in the phase that naturally follows the last one illustrated above, when we would begin for example

pulling out a common superclass for Player and Enemy with a number of shared properties and functionality to minimize redundancy. The upside of the tradeoff, to reiterate, is that splitting the code out from one hacky function into multiple methods, classes, or files improves code readability and separation. That makes it easier to share code with another programmer, or even to simply make sense of it yourself, as the programmer, a week or more later when trying to work on improvements to a particular part. Those fixes are often worth doing, at least on an initial pass, to minimize getting caught up in a tangle later. Another approach I've seen is when people will crazy hack during prototyping on a rapid and flexible platform, like Processing, then once they figure out some of the core gameplay interactions start over fresh in reimplementing that gameplay using native Objective-C code for

iPad, finding ways to trick Unity into behaving the same as the prototype, etc.

There are good reasons why rough drafts don't (and shouldn't!) look like final drafts. There are likewise absolutely good reasons for why later drafts need to look less and less like the rough draft as the work begins to solidify.

Learning and becoming comfortable with a few different practices and processes for managing this transition can help you be the kind of developer that can see through your own original designs, rather than being stuck at either the hacky end as a gameplay designer unable to logically divide the work to scale beyond a prototype, or being the sort that's only comfortable working on more robust implementation of concepts figured out by (or cloned off of) another developer that's more comfortable juggling the flexible, messy, and iterative design discovery work.

THIS IS REPEATABLE, ITERATIVE, LAYERED

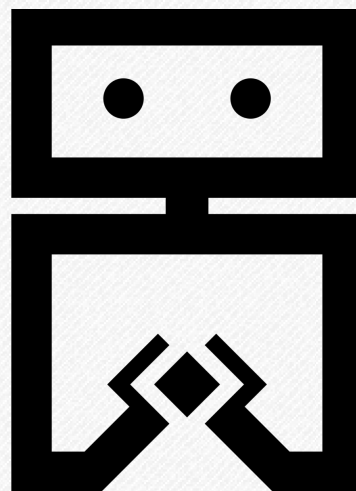
As one final note, I'd like to point out that this is rarely something that happens one time to the entire game's code all at once. That may be the case early on, while it's still a very small core being figured out, however iterations atop of that while exploring potential features can then take on the same pattern shown above: hacking it first in a rough fashion to try out, in order to efficiently figure out what's worth further organizing into the larger code structure to keep and evolve, versus what to trim back out before proceeding.

TRY IT! PUT IDEAS INTO PRACTICE!

Don't just read about videogame development! Put these ideas to work for you. If you're already familiar with programming this doesn't take very long to try - that's kind of the whole point of this method.

BASIC REAL-TIME VIDEOGAME ARTIFICIAL INTELLIGENCE

Many books out there about artificial intelligence are written by AI researchers, and involve have layers of complexity that we don't need for hobby projects. Here's a powerful main idea without all the extra clutter.



SMART IS NOT WHAT PEOPLE THINK

"The question of whether a computer can think is no more interesting than the question of whether a submarine can swim."

-Edsger Dijkstra

There are a lot of films, novels, and comics about artificial intelligence. These present AI as a problem solving, self-preserving intelligence eager to assert its dominion over people.

There are a lot of non-fiction books, internet references, and university classes on artificial

intelligence. These present AI as a pattern-matching, data-mining adaptive algorithm used to process mountains of messy, complex information.

AI in real-time videogames is not about trying to program intelligence. It isn't about making the best decisions, thinking through complex problems, or adaptive learning.

AI in real-time videogames is about creating the impression of intelligence. Generally speaking,

the appearance of intelligence is all that we're going for – it's functionally pretty similar in such narrow domains, and infinitely more within reach than creating "real" intelligence.

What are the qualities that we can appeal to, in order to create the impression of intelligence during gameplay?

SMART IS UNPREDICTABLE

If the "smartest" opening is move A, that move had better not be the opening every time, or the player will just be conditioned to react to that opening move every time. So long as one strategy works repeatedly, what the player is up against doesn't come across as intelligent, just mechanical.

The moment that pattern is broken, or that expectation is violated, the player will project intelligence onto the previously robotic computer entity. "Why did it switch to move B?" The real answer to this question doesn't matter – whether the program looks deeply into the player's eyes

and reads his or her mind, or simply has a random 20% chance of doing something random – the important thing is that the player is forced to stay alert instead of always leaning on a memorized strategy. (Guess which of those two strategies is the most cost and time effective solution? Mind reading, or inserting a line that reads `if((rand()%10)<2)...`)

The 1997 game *Trespasser*, based in the Jurassic Park universe, attempted to create AI that "thought" based on needs around the ecosystem, but the end result was monsters wiggling around looking confused until they charged sideways at the player. By comparison, listen to a young person or non-technical player spending a few minutes in front of *Ms. Pac-Man*, and before long they'll blame the mostly random ghosts for all sorts of devious schemes.

Keep it simple. Throwing in a good dose of randomness at a high level for the enemies is often just

as effective at creating the illusion of intelligence for the player as a more contrived system of reasoning that would take longer to develop.

SMART HAS PRIORITIES

Heuristics – “fuzzy logic” is another term for this – is a way to simplify complex decisions involving many considerations into a decision about decisions.

The concept here is a powerful one, but simple once introduced. Instead of building a massive tangle of if-else conditional statements and boolean logic to determine an AI character’s next move, the goal is to encapsulate what information derived from those variables means.

It’s a form of divide and conquer. To use a business metaphor: how does a top-level executive make informed decisions? By asking each of their advisors and expert employees to offer distilled summaries of the issues they each believe are most important. In the same way, rather than trying to

deal with countless variables all in the same part of code, we want to distill groups of variables into meaningful measures that we can base the high level decisions upon.

SHORT BREAK FOR EXAMPLES

In Burn 2, a real-time PC game involving up to 10 fast-moving AI ships thrusting in a variable gravity environment with procedurally-generated fully destructible terrain (in other words, very little absolute certainty), each AI has a few values that they evaluate independently each frame, and behaviors are determined by which of these values exceeds a priority threshold:

1. Is the ship above minimum safety altitude to avoid ground collision?
2. Is the ship below maximum altitude to ensure room for evasive maneuvers?
3. How imminent is collision with a mountain (laterally) if nothing changes?

4. Is the nearest enemy in firing range? Am I facing them to immediately begin firing?
5. How close is enemy fire? Is one facing me which may be firing, and if so, am I facing them (to fire back) or facing away (to thrust away to safety)?
6. Am I about to collide with an enemy? If so, do I have more health than the enemy? (Collision causes equal amounts of damage to both ships involved)
7. Is my current target low to the ground, with little distance on each side, implying they are entrenched for cover? If so, switch to bombing patterns...

In Ghosts in the Machine, a real-time PC game features 3-5 high speed projectiles ricocheting between 4 independent players (3 of which are AI). The AI keeps track of the following information for decision making:

1. Is a ball heading toward my goal? (forget any other objectives, and protect the goal)
2. Which players currently have a clear shot at my goal, and a ball to fire? (block that angle)
3. Which players do I currently have a clear shot on, and if I have a ball to fire, are they blocking the gap?
4. Is another player currently swamped with lots of incoming projectiles, that would be too overwhelmed to block if I fired now? (fire on them immediately!)
5. Which player has caused my base the most damage? (aggressively hassle that player, to knock them out)
6. Which player has won the most rounds? (target them, so that it won't come down to one on one against them later)

SMART HAS PERSONALITY

By changing which of the above is a priority for each AI player, I was able to produce personality

differences between each of the AI figures (defensive, aggressive, and retaliatory), and by shifting a value between their impatience (premature firing) vs their sound decision making (higher threshold for likelihood of successful shot before firing), I was able to vary any of the 3 personality types from easy/dumb to hard/intelligent.

SMART ACTS DECISIVELY

Moods – or “finite state machines” – can encapsulate a sense of memory, reaction, decisiveness, and thus intelligence. The way this works is that each character keeps a number tracking its state of mind. Is the enemy wandering, alerted, angry, or retreating? Is the enemy grouping together, spreading out, hunting for a health pack, or battling to the death on a kamikaze charge?

SMART GROUPS HAVE EMERGENT TEAMWORK

For an interactive demonstration of mixing randomness and determinism in AI that can be played quickly online, check out

my experimental project BeeDifferent:

http://interactionartist.com/classic/gameloader.php?GAME_NAME=BeeDifferent

Experiment with the slider below the game to see how enemies which all behave optimally are trivially outsmarted, enemies which all behave randomly are also easily outsmarted, but by using an even mix of the two, the effect is the have some spread out as mines while others force the player to run through said mines.

SMART IS VERBOSE

Telegraphing the enemy’s current state, and/or the transition into that state change, also goes a long way in creating the illusion of intelligence (though this shouldn’t be too repetitive). The enemies turning red with anger in Bubble Bobble declares their current mood clearly. Metal Gear Solid bad guys get exclamation points over their heads, and enemies in No One Lives Forever and Thief (both of which were widely praised in their time for their otherwise

normal AI, thanks to this single reason) announced things along the lines of “I can’t find her!” and “I know you’re here!” and asking their comrades things like, “Did you hear something?”

SMART BUILDS UPON SMART

Big games – whether a First Person Shooter, Third Person Adventure, or Real-Time Strategy – invite some other AI needs like pathfinding. FPS and TPA games often include hand-placed nodes in the world giving AI clues for where doors are, where the center of hallways/rooms are, and where the best cover points are to kneel behind. RTS games can involve a bit of clustering (basics of facial recognition algorithms) to assess areas of defensive weakness. Basic vector geometry and dot products can be used to ascertain line of sight between characters (including trivially checking a character’s facing), and trig (especially atan2) are useful in finding angles to shoot in, look at, run toward, or flee from.

Puzzle games are obviously a whole different can of worms, depending entirely upon the game’s structure.

All that said, much of the real-time AI in most games still tracks back to (a.) creating the illusion of impatience and intelligence by being unpredictable (b.) reducing groups of variables into easily prioritized decision points (c.) committing decisively to display a range of possible behaviors and (d.) unsubtle announcements of the current behavior or moments of mood switching.

SMART CODE IS SIMPLE

On the technical side, the AI “mood” (finite state machines) can be tracked simply by keeping two more integers per enemy. The numbers are used by setting one of the integers to an enumerated value corresponding to behavior #1 vs behavior #4 (etc.), and the other tracks how long the current behavior should continue before being reassessed. For example:


```

enum {
    MOOD_WANDER, // same as "#define MOOD_WANDER 0"
    MOOD_CHASE,  // same as "#define MOOD_CHASE 1"
    MOOD_MIMIC,
    MOOD_FLEE,
    MOOD_NUM // useful for randomizing mood
};

/* the larger the first number (200), the more
   decisive the enemy seems; the larger the second
   number (150), the less predictable the enemy seems */

#define NEW_MOOD_TIME (200+(rand()%150))

class Badguy_typ {
    // ... skipping class variables for brevity ...

    int aiMood;
    int cyclesTilMoodReconsidered;

    Badguy_typ() {
        aiMood = MOOD_WANDER; // begin unalerted
        cyclesTilMoodReconsidered = NEW_MOOD_TIME;
        // other character initialization goes here
    }

    void Move() {
        if(--cyclesTilMoodReconsidered <= 0) {
            // random new mood
            aiMood = (rand() % MOOD_NUM);
            // random new amount of time till change
            cyclesTilMoodReconsidered = NEW_MOOD_TIME;
        }

        /* cyclesTilMoodReconsidered is only
           intended as timeout, so that left on
           their own, unprovoked, enemies will
           seem unstable, curious, and impatient,
           like they have minds of their own. */

        /* Within the switch statement below,
           it's fine to switch aiMood and reset
           cyclesTilMoodReconsidered even if it
           isn't time for a mood change yet. For
           example, if the enemy is within a certain
           range from the player, especially if line
           of sight is established, then just switch
           "aiMood = MOOD_CHASE;" */

        switch(aiMood) {
            case MOOD_WANDER:
                // enemy is wandering, code here
                break;
            case MOOD_CHASE:
                // enemy aggressively chasing player
                break;
            case MOOD_MIMIC:
                // enemy is copying player movements
                /* this is very easy to implement but
                   has the nice effect of giving the
                   player a temporary sense of control,
                   making it so that when this character
                   switches to a different mood there's a
                   good chance of catching the player
                   off guard */
                break;
            case MOOD_FLEE:
                // enemy running away from the player
                break;
        }
    }

    // ... skipping other class methods for brevity ...
}

```

This sample code snippet pretty well describes the AI for most of my first PC games, including Pac-Deli, and Swarm. It's not going to appear intelligent in a one-on-one fight, but when the player is up against 3-8 enemies operating independently in this fashion, the player begins to interpret the emergent patterns of their mostly random (unpredictable!) behavior as teamwork, plotting, and strategy. In Pac-Deli I gave each ghost color a significant bias toward 1 of the 4 behaviors - half the time using their "personality" and half the time switching to a completely random choice - and this created the impression that one was more confused, one was more aggressive, one was intent on being tricky.

WE'RE NOT TRYING TO BEAT KASPAROV

Ultimately, it's okay if the AI isn't the smartest thing on earth. If the player outsmarts it, they'll feel like a champ. You don't need to prove to players that a computer can

process and respond to
information faster than they can.

As Ian Davis, the founder of Mad
Doc Games told me, "The goal of
videogame AI should be to put up
a fair, convincing fight, and
eventually lose."

QUICK AND DIRTY BACKUPS

When a hard drive fails, or a laptop gets stolen, you can reinstall most programs from a disc, or the cloud... but not the ones you were in the middle of creating! You can't prevent accidents. You CAN minimize the loss.



I received a number of questions from a reader about version control for hobby videogame projects. My responses follow.

Q: HOW BIG DOES A PROJECT NEED TO BE TO BENEFIT FROM VERSION CONTROL?

A: In the laziest, simplest form, I think even a solo game jam can benefit from some crude form of version control. As for “real” version control, I tend to only bother if at least 3 programmers are involved with the project, and the project is expected to take at least a few weeks or more.

Q: IF I USE VERSION CONTROL, WHAT SHOULD I USE? GIT, MERCURIAL, OR SOMETHING ELSE?

A: My opinions on this are likely out of date, so I'd suggest investigating the options to see what's being said about these newer options. For large-scale commercial development I've used Perforce, and for smaller independent student teams I've only used svn.

Q: WHAT CLIENT SHOULD I USE? GITHUB, BITBUCKET, OR SOMETHING ELSE?

A: Likewise, my tastes here may be slightly outdated, but they still work. I used TortoiseSVN when I

was in Windows, and just handled svn via command-line when I was in Mac. I've witnessed a number of experienced programmers, myself included unfortunately, bind up a bit when running into certain conflict or folder mismatch issues via command-line using svn, and a good client would also probably go a long way in simplifying that experience. Having a good file comparison tool is important to spot diffs in a conflict, and that's often part of the client with Perforce or Tortoise, although on Mac I used a program called FileMerge which if I remember correctly was a tool application bundled with XCode.

Be aware that some online services that provide hosting servers or other niceties may come with strings attached, such as requiring that your program is open source, etc. That may sound fine, but even if you're keen on the spirit of it, that can sometimes drag out a project needlessly by making someone overly self-

conscious of how well organized and documented their code is. When code isn't open source we can occasionally resort to hackery to get things done in special cases without getting self-conscious about it, making conscious tradeoffs to optimize our development time in that case (as opposed to, say, making tradeoffs in readability to optimize code execution time). Of course, if you're a full-time software engineer and used to or interested in joining larger team environments, for that same reason open source might be a constructive exercise, in which case such licensing implications may not be a concern one way or the other.

Q: HOW OFTEN SHOULD I UPLOAD A NEW VERSION?

A: I'm a proponent of trying to work on one relatively isolated, bite-sized (1-4 hours of work) feature at a time. I'll tend to commit:

- Each time I've completed and tied up a functional chunk, generally amounting to several hundred lines of code or less. This can even become a nice part of the ritual, punctuating a finished task, like scratching something off a todo list.

- Whenever I finish solving a tricky or creative code issue, or finish a tuning or balancing pass I'm pleased with, that I'm not confident I could easily reproduce a second time.

- Upon finishing something monotonous that I would not want to do again. This mostly happens when doing some type of hand-adjustment to assets, for example tweaking spacing offsets for every character of a medium or high resolution bitmap font, or renaming several folders of sound and image files to fit an improved naming convention.

It's important to only check in code that's ready to compile and run though without interrupting

other features or debugger output (i.e. for log or trace statements: clean up, remove, or make toggled on/off in code that defaults to off, before committing to svn), or it'll throw off other programmers on the project, including future you. If that code is otherwise incomplete, then for now make sure it has been safely isolated from other code still in use, flagged with a short comment at the top indicating it's not ready for use. Be careful to not break the build, hog the debug output, or degrade performance with a half-finished check-in that will leave others wondering whether their local changes caused it.

Q: IS VERSION CONTROL WORTH IT IF YOU ARE THE ONLY PROGRAMMER ON THE PROJECT?

A: Here's what I've been doing for the past 5 or 6 years when I'm the only programmer on the project (which is very often the case for my hobby projects, even when there's a team of people involved with assets), it's a habit I picked

up from one of my longtime collaborators John Nesky:

- Every night, when I'm at a good stopping point, I zip up the project's folder. I save most of the old ones of this, or at least 1 per week, and it makes it easy for me to dig my way "back out" if I find that I've coded myself into a corner or introduced some nightmarish bug I'm having trouble undoing. Sometimes it's also fun when I'm done with a project to look back on old zip files to see what the project was like earlier in development – what didn't make the final cut, how much rougher it looked earlier, etc.

- At least once a week, I copy the zipped folder to a location online, like a folder on my FTP server not visible to the outside world. A local external hard drive for back ups is handy, but not sufficient. Nowadays I just copy the zip with dated filename into a special folder that I keep on Dropbox specifically for development backup, and it's automatically

copied to the cloud. The idea of keeping a version of the full source at a remote location is that no matter whatever disaster could happen locally – backpack with my laptop in it falls into a deep puddle, catastrophic hard drive failure, my home gets broken into and my computer and backup drive both get stolen, home burns down, whatever – then at least I have something to pick back up from.

Loss would still be inconvenient, it would still be expensive, it would still be lame, but things can be replaced with money, whereas lost game development work is simply lost time, a huge loss to morale, and if there are other people on your team or a business counting on you to finish, not tending to this properly can make your or my misfortune spread to also become their misfortune. It's way too simple to back up a project remotely to have any excuse to ever completely lose a project's

code and source/raw/PSD art files.

It's not your fault if something goes wrong and a hard drive utterly fails, but it is your fault if that happens and you've not taken any steps to mitigate that loss.

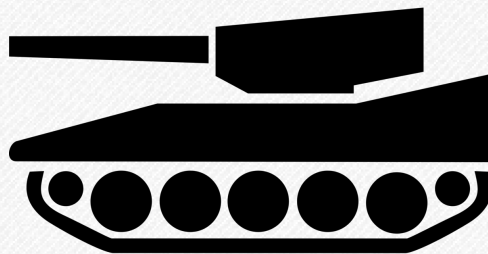
If a project is tiny enough and you're at a game jam, and don't want to set up a Dropbox (but you really should, it's free and easy to do), at least e-mail yourself a zipped copy of the source every so often. If you don't have internet access for some reason, occasionally copy that latest zip to a USB stick so it's not all one the same physical point of potential failure.

It's not either-or, though. There's a continuum of source control between having the technical infrastructure of a massive studio, which needs a fully staffed IT department to keep it running, compared to the resources of a hobbyist or student trying to moonlight. While it's important to

not become bogged down with sorting through non-game technical troubles, it would be reckless to have all of the work invested into a project living only in a single directory having one version on one machine. That's terrifying. At the very least, make a new zip of the directory after a good night's work on it, and occasionally toss a dated version of that file onto a server in a remote location. If the project is so large, distributed, or complicated that using that method isn't practical, then it's time to look into setting up some real source control, whether svn, git, or one of the other solutions out there.

STEPS IN PROGRAMMING A SIMPLE REALTIME STRATEGY GAME

This student's question helped me highlight the common layered approach to game programming. Start by making something basic, then at each step revisit adding new functionality and features to your work.



Q: I've been wanting to do a military strategy game. I learned C++ in high school. I made a few text-based games, but nothing major. Any starter tips?

A: If we were to make a demo or proof-of-concept version of the military strategy game that you have in mind – i.e. pretty much the bare minimum of functionality and assets to demonstrate what it does and how it goes it – what would be involved? For starters, that should be the target, rather than a full game. If the core functionality of a “demo” works

(one level, a couple of enemy types, basic winning and losing), then you can use that as a foundation to build the rest of the game; if that proves out of reach, then you'll figure that out far sooner, and be able to adjust plans or switch projects before getting in too deep.

As one possible starting outline, in whichever language, API, or environment you choose to do it in (Unity for cross platform, C++ with Allegro or SDL for high

performance downloadable, AS3 for web, etc.)

STEP 1: MINIMAL FOUNDATION

Get the mouse showing up, either as a default cursor or by drawing a small circle where the mouse coordinates are. Create two numbers, x and y, to store a soldier's position, and draw a colored rectangle on screen at that coordinate representing the soldier. Before moving on, set it up so that clicking close enough to the soldier kills him – this can be as simple as a distance check between mouse cursor and the army man each time the mouse clicks, setting a “dead” boolean flag to true which draws him a different color. The distance check to the soldier should be written as a separate function that uses the Pythagorean theorem to return a distance for numbers or coordinates given as parameters. (If you are going the Unity route, some translation may be appropriate, such as rectangular prisms instead of squares, and

checking raycast collision against a bounding box a bit larger than the unit rather than doing distance checks to detect click proximity.)

STEP 2: GAME LOOP

Create a game loop so that action can happen 20-60 times per second, rather than only when the mouse clicks. If programming in a local native application or prototyping environment, this can be done by making a boolean “gameRunning” set to true, and wrapping the core of the game (drawing, input handling) in a while(gameRunning) loop. If the user either presses a key (Escape perhaps) or clicks in the top-right corner of the screen (coordinate comparisons), set gameRunning to false to quit the program. If you are in a newer environment, this is often done instead of using a loop by setting up a timer to call a main logic and rendering function 20-60 times per second, which is where your game code can live. In Unity this is set up automatically in the

form of the Update() tick functions built in to the scripting.

STEP 3: TARGET DESTINATION

Have the soldier keep track of a target destination by creating two new numbers, tx and ty (standing for target x and target y). Update the target destination to where the mouse last clicked. In each frame in the game's logic, if the soldier's target destination is more than some distance from his current location, have him move closer to his destination. To fudge this, just do a straightforward:

```
if(x < tx) { x++; } // left of it? move right.
if(x > tx) { x--; } // right of it? move left.
if(y < ty) { y++; } // above it? move down.
if(y > ty) { y--; } // below it? move up.

/* Remember: an increase in y is down, by default, for most game programming.
If using Unity this would need to be done differently, among other things using x and z as the
horizontal coordinates instead of x and y */
```

If you want to be fancy, use atan2() to find the angle from (x,y) to (tx,ty), then use sin and cos of that angle times a move speed or a bit of vector math to move the soldier directly and evenly toward the destination. If this seems correctly coded but is acting strange, double check to ensure that you're

using float precision numbers for coordinates, rather than the whole number "int".

STEP 4: REFACTOR INTO OBJECT

We've previously been keeping track of the soldier's position and destination with variables of ambiguous scope or organization, presumably global. Wrap up the soldier's x, y, tx, and ty in a new struct or class, and update the code to use the object's values (they can be left public for now).

STEP 5: MULTIPLE SOLDIERS IN ARRAY

Refactor that code so that instead of having one soldier instance, you have an array or list of soldiers in the world starting in various areas. Keeping track of a current and target position for each should be automatic given an array of the struct or class defined in the previous step. Also add a new number: an index indicating which soldier in the array is "selected" (set to -1 by default, to indicate that no soldier is selected) – as always for Unity this will require some adaptation, keeping

a GameObject or Array of GameObjects as a handle on which unit(s) the player selects which would be null or empty by default and between selection. Adjust the previous “click on to kill” code so that it sets the selected value to that soldier’s index – a brute force iteration through the entire array comparing each soldier’s distance to the mouse is fine at this scale. If no soldier is found close enough to the mouse to change the selection index, set the currently selected soldier’s tx and ty to where the mouse clicked.

STEP 6: DIVIDE ARMIES

Give the soldiers one extra integer: army number. Set to 0 for half of them, 1 for the other half, and draw them as different colors. Only let the player select units matching a particular army number.

STEP 7: TARGETING AND HEALTH

Give the soldier class/struct two more integers to apply to all soldiers: a killTarget index

(GameObject reference if Unity) where another soldier’s index can be stored (which can be used to change tx,ty coordinates if beyond firing range) and hitPoints integer that starts at 3 and goes down whenever hurt. Add a constant number value (or if using C/C++ a #define will work fine) to centralize your definition for gun range. Have the code draw a line from any soldier with a non -1 killTarget to the position of the enemy having the index they’re targeting.

STEP 8: ATTACK COMMAND

If an enemy soldier is clicked, instead of selecting that soldier, set the target of the currently selected friendly soldier to that enemy. Use random countdown timers between shots fired (maybe every 100-300 ms?) to randomize who wins, and/or give a probability of missing (if(rand() % 3 != 0) gives them a 1 in 3 chance, etc.) and depict the missing, firing, or hitting in some way like flashing circles, sapping health from soldiers that take damage.

STEP 9: UNIT DEFEAT

If a soldier's target is defeated, either assign them a new living target at random or await player orders.

STEP 10: COMPUTER CONTROLLED ENEMY

Make AI by having countdown timers either per enemy soldier or per enemy army between discrete moves, which could consist semi-randomly of one of three things: move a soldier to a new spot, move two soldiers closer together, target a random or the nearest player soldier, or team up with a fellow soldier by selecting the same target as his nearest teammate is targeting. Tune those probabilities and timers until it's reasonably fair.

STEP 11: NON-INFANTRY UNITS

If you've made it this far – congratulations! It's an infantry war game now. A simple `isUnitType` integer added to the soldier class or struct could correspond to an enumerated unit type (`TYPE_TANK`, `TYPE_PLANE`, `TYPE_ARTY`, `TYPE_GRENADIER`, etc.)

4



Even once you know what to do, you've still got to put it into practice before that knowledge can do you any good. Becoming good at anything requires practice. Let's look at some ways to start getting practice sooner rather than later.

GET MOTIVATED

STOP TRYING TO LEARN EVERYTHING BEFORE STARTING

Knowing about something isn't the same as knowing how to do it. The former comes from reading and discussion, the latter requires practice and experience. When you know enough to start... start!



Don't wait until you know everything there is to know before starting.

Learn just enough to get started. That includes having a rough idea of a realistic scope for a first project, so that you don't wind up lost on a fool's errand.

Then get started. Get your development environment set up and compiling empty, test, placeholder, or example code. Just get anything on the screen that runs, meaning an .exe if

you're programming for Windows, a .swf if you're making a web game in ActionScript 3, etc.

At this point, you're already ahead of an untold number of people that have only ever thought about videogame development, but have been too caught up in waiting for the perfect idea, or trying hopelessly to fill their brains with everything ever written and said about it before actually getting started. So far so good.

Find a way in your programming language or environment to load an image file and get it on the screen, to respond to keyboard and mouse or whatever input is needed, to load and play back sound effect files and looping music. Depending on the platform, picking a library may be helpful for this (ex. SFML, SDL, Allegro, or XNA, if in C/C++), or this may be functionality built into what you're working with (Unity). Often the easiest way to get to this point is to just find some simple example code that already does these things, then twiddle settings in your development environment until you can compile and run it as expected.

Does the player character need to move like a truck? A spaceship? A tank? Mario? Get the input to move the player's graphic (probably an unanimated rough draft of the graphic at this point, concerned only with basic appearance and scale on screen)

moving in the way that it ought to move.

Get the level structure put together to position visuals and handle basic collision against the player. Save/load the level structure in some practical format – never mind getting caught up on the theoretically optimal compression of that data, as level files are virtually always tiny, and if that becomes a problem later, cross that bridge when you get there. Level files initially saved in ASCII can make debugging significantly easier anyhow. Is the game world based on freestanding obstacles? Grid-based tile collision? If the camera needs to pan, add some offsets to the player draw position and level draw origin to achieve scrolling, or a global transform if you're in 3D or using hardware acceleration, and update those offsets to move the view based on player position.

Add enemies (if needed), items (if needed), trigger puzzle elements (if needed), special powers (if

needed), ammo limits (if needed). Add nothing that isn't needed. If you're not sure what else the basic engine might need – and I'm only using the word “engine” here only in the most minimal sense, meaning the game's core code, not the try-to-support-every-game-imaginable Titanic-seeking-an-iceberg undertaking – start putting together playable level content and see what additional features or process improvements you find yourself wanting while doing that.

The game may not need anything else.

If and when a situation arises for which you're not sure how to proceed without digging and experimenting for solutions, that's fantastic, welcome to real programming.

If and when a situation arises for which you have to make a tradeoff between burning an uncertain amount of time on figuring out something tricky, or working out

how to cut that feature from the design without the game completely falling apart, that's super cool, and welcome to production.

Once you finish the game and people don't feel about it the way you hoped they would, first off: that's awesome, because hey, you finished a videogame.

Congratulations! Next time you'll be positioned to make incrementally more informed tradeoffs with consideration for implementation realities, production compromises, and potential impact on user experience. It's only at this point that someone is beginning to really do videogame design, as opposed to talking about videogames, reading about programming, or cloning someone else's work as an exercise. Note too that design of a full videogame is different than level design. Though the two are sometimes conflated, since they are often done by the same people, when

done in isolation from other issues and options about a game's development, level design is another field of content creation, another skilled technician's craft like animation, dialog writing, or sound editing. Those are hard things to do, deserving of respect and worth doing well, but they are not videogame design. Designing a videogame is different than designing for a videogame.

Learning is great. Books are important. Some amount of learning does need to happen before starting, but the amount of learning needed tends to be far less than people seem to assume. Learning, for this type of material, has to be situated in a context, demonstrable, useable, and practical. Contrary to the lay consumer impression that making something is a mere variation on being able to judge something, a maker has a fundamentally different set of concerns and practices than a critic.

Roger Ebert was brilliant, but wasn't suited to making a decent film. (Not hypothetical - see: *Beyond the Valley of the Dolls*.)

If your goal is to be a critic: spend all day studying the form and discussing it. If your goal is to be a maker: start making things, keep making things, and finish making things. Being reflective on practice can be helpful, but that requires actual practice to be reflective about.

Learning everything there is to know about programming or videogame development before starting is impossible. It simply won't – can't – all fit inside one head at the same time. Even if that information somehow could be learned, memorized in the abstract, detached, and rote sense, it would not be of any use without experience working through real problems with it.

This is not a decision between learning versus doing. What I am proposing is that, within this

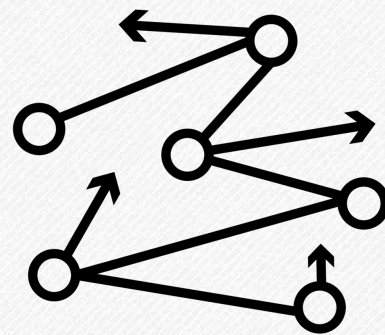
context, learning without doing isn't really learning. Meanwhile doing without learning is impossible, because doing will demand learning.

If you make some wrong assumptions early on, they're often easily corrected. That's the nature of digital stuff. We're not laying railroads or building skyscrapers, we're not cutting someone open while they're under anesthetic – we are dealing in digital text and other easily modified, easily backed-up files. The rework time is nothing compared to the vast but invisible damage from never starting, waiting an infinite amount of time until everything knowable is known. That rework is even when the best learning happens, because the desire to not lose that amount time on the next attempt will help lead to a deeper understanding of what caused it, so that it might later be avoided.

Be wary of excessive preparation serving as a disguise for procrastination.

"OVERCOMPLICATING EVERYTHING"

While the previous section distinguishes “know about” from “know how,” here I distinguish “know how” from actually doing what you know how to do. I turn to Arnie, everyone’s favorite action hero, for inspiration.



Arnold Schwarzenegger is an incredibly inspiring person. At 20 years old, he won Mr. Universe as a body builder. For many of us, we could just look at that achievement and see that as the likely high point, figuring that might come to define our identity and life path. No, that wasn't enough though. He started a successful mail order business based on his recognition from body building. Here again, many people might see that as a stopping point – he became a

millionaire well before he was a movie star. Still not enough, the guy picked up his life and moved across the world, becoming an actor. And, at least so I once read, his first agents advised him to change his unpronounceable, unspellable, foreign-sounding last name, encouraged him to hit hard the speech therapy to drop his accent so that his voice could sound more American, going so far as asking Arnold to get plastic surgery to make his chin and face less boxy, because it did not fit the

image at the time of how a handsome Hollywood movie star should look.

No. No. No. He found someone else to be his agent, because he was determined to succeed on his terms, with his name, with his voice, with his face, and he of course did. With flying colors. He helped define a whole new type of action hero, an image that future Hollywood agents could use as a measuring stick when encouraging new Hollywood immigrants to... maybe not change too much. Again like becoming Mr. Universe, and again like creating a successful mail-order business, anyone would have understood at this point had he simply accepted his identity within the world as a movie star, a beloved and extraordinarily successful one, finding a way to be satisfied with that.

He become governor of a state in a country other than the one in which he was born. Not just any state: a state that has 4.5 times

the population of his home country and 5.5 times the GDP of his home country. Borrowed directly from the Wikipedia entry on California Economics:

“California’s GDP is larger than that of all but 8 countries in dollar terms (the United States, China, Japan, Germany, France, Brazil, the United Kingdom, and Italy).” He became head of the executive branch of the 9th largest economy in the world, in a place he immigrated to in his 20s, and for eight solid years because he won the re-election by a wide margin (if it started even partly as a joke, clearly he succeeded in quickly taking it quite seriously).

Hoo-oo-oo-leeeeeey schnike-ees. It’s hard to really wrap our minds around the thought of becoming a mayor in a small city we grew up in. Oh, and he started a restaurant, and probably some other impressive things, but I suspect you’re getting the point. Schwarzenegger is basically like Tony Danza on steroids. Okay,

formerly on steroids. (Tony Danza, you see, has also done a lot of stuff, not letting his early career determine or limit his identity and interest in broadening his life experiences.)

This is all background, mere set up, though I suppose it has a bit of a valuable message in its own way. The reason I bring up Schwarzenegger is because periodically, he participates in an open online forum about weight lifting, offering advice and answering questions. And recently an exchange took place that I think is worth calling attention to here:

Q (Snowman24): When trying to cut weight after a bulk, what was your best method? In terms of cardio, diet, etc? Do you believe in ketosis at all?

A (GovSchwarzenegger): Pretty simple... I would add in extra cardio – running on the beach, swimming, and bicycling. I cut out bread, pasta and desserts. It

definitely wasn't rocket science, but it worked.

Random observer (rainman1): I'm getting the distinct impression we [have] been overcomplicating everything.

Glorious. Here's a massive, highly active discussion forum where people come up with, deeply research, and argue over complex diet and workout schemes to optimize their gains, asking for advice on how to lose weight based on the lifelong experiences of Mr. Universe, a 7-time Mr. Olympia winner, a man that can still achieve heroic muscular fitness now in his mid-60's for continued action movie cameos, and his answer, basically: exercise more, and ease up on carbs.

Pure gold!

It's maybe just a little too easy for some slightly geeky people (if you're reading this, I trust that you're at least a little bit geeky. I'm geeky too) to judge some meathead, weightlifting jocks (I

was sort of one of those, too) for overlooking the obvious, making it seem more confusing or difficult than it needs to be. However, we videogame developers routinely fall into doing that exact same thing to ourselves.

We often know what we need to do next. Instead of just doing it, we get into a long-winded philosophical argument, tangential discussions about directions that we definitely are not going to go in, or bury our faces in Google trying to find everything that there is to know about this particular decision before we just, 99% of the time, finally get around to doing what we were going to do in the first place, in the same way that we were going to do it in the first place.

Generally we'll only actually try an alternative approach if that first attempt catastrophically fails us. Even then we're very likely to attempt the next most logical, convenient, or familiar thing that we can find practical information

for. This isn't being lazy; this is how things happen at all at our scale. Given our small (or solo!) team sizes and the sheer amount of code, data, content, information, and decisions that we're trying to juggle to make videogames, it's often smart, or at least sensible, to be efficiency-oriented most of the time. Then we can better make it count when we deliberately do something specifically contrary to conventions and expectations for a meaningful reason.

In other words, assuming we're on track to realistically get a lot done, what we do typically has far less to do with all that extra discussion and investigation, and much more to do with what arises from acting on our ideas and reacting dynamically if they don't come together as expected. There's often simply no way to know whether they can come together as expected until we've tried acting on them.

It's almost as though, somewhere deep in the recesses of our minds, we think that if only we could know enough about the problem at hand, one morning we'll wake up and the work will be done. My generation grew up hearing from G.I. Joe that "Knowing is half the battle" but maybe didn't hear often enough that the other half consists of taking action based on that knowledge. It's not as catchy, I know.

No amount of accumulated knowledge is going to substitute typing in the next characters that need to be typed, creating the files that need to be created, doing the bit of code refactoring that may be necessary to move forward on the project without tripping over bugs leftover from partial implementation of ideas tried but abandoned because they did not work. (That's just part of the process. Again, generally speaking, that hasty implementation is often the only and most efficient way to figure

out what ideas are or aren't working. When they aren't working, abandoning them is far smarter than forcing yourself to use them simply because the work on them is partly done.)

Here's another hard fact, true for basically every developer that I have ever met, myself included, which I hope can be digested constructively here as a challenge to your skills and intellect: there are a great many things that you think you know how to do, but have never attempted, and in all likelihood you don't really know how to do those things. At best you know how to start, and have confidence that you can figure it out from there, but that vague potential is very different from having done it. Even understanding how individual parts work is not at all the same as knowing how and why they fit together, let alone being able to assemble them. Once we actually attempt to do the things that we only thought we understood, we

quickly discover all kinds of lovely nuances and subtle complexities that weren't apparent on first glance. The upside of course, is that by the time we finish the attempt, we really will understand exactly what's involved in making it work. It'll become one more tool in our conceptual or practical toolbelt for future use, having shown ourselves that it's something we know how to do.

Note, however, that to “know how to do something” has way more to do with the “do” than it has to do with “know” – strange, right? If I ask whether you know how to do a skateboarding trick, I don't mean do you understand what the stunt is. I mean, can you do it? If I need heart surgery, I need a surgeon that “knows how to do” the operation, by which I mean someone that can do much more than just describe the procedure. If you haven't made a platformer game before, you maybe have all kinds of ideas of what you could look up or try first to get started

with doing so, but strictly speaking, you really don't know how to do it, even if you just finished reading a book on the subject.

To people in our family – and to ourselves – we tend to look like our potential, what we seem capable of doing. To friends, we may be thought of from how we seem in the present: less pressure about the future, and not worried about the past. However, to strangers, literally everyone else in the world, our abilities and our character are judged only by our past, what we have already done. So outside of the dozen or fewer people that know us best, if we haven't done it, as far as anyone else is concerned we can't, regardless of what or how much we “know” inside. There is only one thing that can change their minds about whether you can do it, and that's going through with actually doing it. Strangers can be very hard to persuade, but this

turns out to be an extremely effective way of doing it.

Someone could spend years reading about how to do something, then be put in a situation to do it, but then be unable to translate that reading into doing. Meanwhile if someone else has actually done it, we know that we can generally count on them to be able to do it in the future, or benefit from real takeaways learned as part of that experience.

I'm clearly not opposed to books or book learning. I love reading, and I read a ton. Knowledge is great! All I'm trying to stress here is that especially within the domain of a craft, knowledge alone cannot be a substitute for experience of actually putting things together. Here's the crazy part: actually putting things together can in many ways be a lot less complicated than trying to study or talk coherently about them. I'm not saying that doing it isn't challenging, but what I am

saying is that at the very least, doing it is a different kind of challenge than getting endlessly distracted by learning infinitely more detail about the matter at hand.

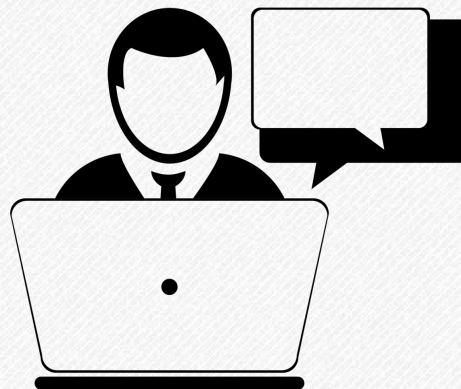
Sometimes we need to – get to! – play the role of grand architect on our projects, but for the vast majority of the development time, we actually just need to be our own construction crew laying bricks, nailing boards, taking measurements, and lifting things into position. Even if you're the kind of person that's capable of governing the 9th largest economy in the world, there's still a time and place in life for simply being a brick layer.

If you feel like you or your work may be hurting from falling into this pattern, and I know a lot of people are affected by this, having been “overcomplicating everything” day after day, I urge you to catch and stop yourself next time this starts. Just get back to the compiler and do the work.

The thoughts you had previously about where this project was headed are still just as valid, though of course even if you're having second thoughts there is no better way to prove to yourself whether the idea works than to simply try it and see.

THINK BY BUILDING, BUILD TO ANSWER QUESTIONS

The act of creation can serve many purposes. Wanting the end result to exist is but one of many reasons. The process of making something can lead us to change our minds, or to refine how we think.



A musician is more likely to dream up new songs by strumming on the guitar than by writing notes on the page. A chef is more likely to invent a new recipe by trying a bold variation on an otherwise known formula – while actively preparing the invented dish, not while sitting in the park with a pen and a notepad. A painter, no doubt, benefits by investing some mental energy in deciding on subject or approach, but I think that the genius of Mona Lisa

happened while standing at the canvas with paint.

I'm guessing at the above, since I'm not a musician, chef, or painter. However I can say for certain that as a videogame developer, I think by strumming on my guitar, trying different spices in the kitchen, and mixing on the canvas.

When we're halfway into developing a game, we think differently about it than we do when we're just getting started or

nearly done. The same is true at 25% through, or 75% through (which often turns out to actually be only 25% through). While building something, we're always at an intersection of "How do I address this immediate challenge?" and "How can I keep this on track to hone in on a coherent, complete result?" Yet before initiating building, the tendency is to think in vague, incoherent, daydreaming ways about the project without either of those helpful grounding questions in mind, because there's nothing tangible yet to pivot on.

Of course, it's valuable to foresee and steer away from potential dead-ends, to have some clear initial direction in mind, and to have a sense for how long a given project might take. I'm not anti-planning. I only mean to suggest treating any such plan as highly tentative – as little more than evidence to yourself and others on the team that there is at least one

path that can lead to coherent completion.

The natural objection to building too soon is a fear that doing so sets too much in stone. That objection assumes the historical approach required of massive physical projects like ships and skyscrapers: extensive planning, followed by huge costs of building. Since the cost of building (and more so, the cost of undoing) in software is significantly lower than in battleships or buildings, I'm suggesting that development is part of the planning process, and seamlessly carries into the production process. Build to clear up uncertainty in the planning, to narrow down frayed possibilities, to work out a plan based in the reality of gameplay with situated proof that A works well and B doesn't hold together.

If a feature isn't working out, ditch it. If a level is bad no matter how it's reworked, lose it. If something was originally planned but what's currently working seems to work

just great without that something, consider forgetting that something and the extra complication it would invite – though if you have the time, build it, and rip it back out if it's not an improvement.

When the built gameplay gets pulled back out, it isn't a loss. It was exchanged for new information that could not be obtained any other way. It was a sacrifice made in exchange for conviction that the better course is to go without that surrendered part.

When actively developing, it becomes clear that there's a much better destination en route or accessible by a short detour. Remember though that the goal isn't to reach the originally planned destination, but to make something worth making. If the original goal's role turned out to be getting you close enough to spot that alternative outcome and achieve it, go for it.

It's okay to make a few not-as-great games, especially if it means that you feel comfortable taking a few chances, trying out a range of new ideas, and working with a variety of different people or influences. However, you can give each project its best chances of succeeding by being bold about cutting out the not-as-great features and the not-as-great content. The flexibility to do that kind of cutting only comes from building to test ideas along the way, beginning with making plans.

Building isn't just for final decisions – it's also for arriving at better decisions. Don't just build to keep what's built; build to think, and build to answer questions.

YOUR ATTITUDE MATTERS EVEN WHEN WORKING ALONE

Attitude is obviously a factor with a team. Issues come in many forms: rudeness, pessimism, unwillingness to adapt, etc. Less obvious is that attitude is just as crucial a factor when working alone.



THE WHOLE TEAM'S PROBLEM

If you're the only person working on the game – as is so often the case for beginning practice, small side projects, and artistic games for personal expression – then your mood is the mood of the entire development team.

If a game's entire development team is feeling angry, is letting frustration overwhelm them, or simply stopped believing in the worth of what they were creating, we'd expect that to drag down the game being made.

If you're working alone and are being unnecessarily hard for yourself to work with, that has to be remedied.

This might seem like a peculiar way of looking at it, but isn't it much stranger to instead assume that the mood of the entire development team has no bearing on the work getting done?

ATTITUDE'S A KEY INGREDIENT

Regardless of how much talent, know-how, and time is available: if your attitude's out of whack, the game won't get done. If your

attitude's a bit better, the game may get done, but not as well nor as soon as it could be. If your attitude's well-tuned, you can find yourself getting more done, at higher quality, and feeling better in the process.

On a team someone else is likely notice these kinds of issues and raise them as concerns to be addressed. What makes this sort of problem especially dangerous for lone developers is that it's much more difficult to catch these in ourselves than it is for us to observe when they're happening with others. The damage is still there but there's no other perspective involved to help spot it.

Attitude is a broad word – on its own it's no more specific than “disease.” Attitude issues are not confined to one kind of ailment, one set of symptoms, nor can it be fixed by one common solution.

If you feel like your body and mind are putting up resistance to

what you're trying to get done, that's a signal that maybe something needs to change.

WHEN THE ISSUE IS ENERGY

The solution might be getting on a more regular sleep schedule, getting out of the apartment more, starting and sticking to a light exercise routine, eating better, or reaching out more frequently to some other people just to talk.

These may sound unrelated to game development, but keeping what's sitting in front of the keyboard properly functioning is clearly at least as important as ensuring that you've got the right hardware and software properly functioning behind it.

WHEN THE PROBLEM IS MOTIVATION

Find ways to reward yourself. Try listening to a different kind of music (more upbeat? more energetic? more nostalgic? different things work for different people). Rethink whether your development schedule might be adjusted to keep your morale higher through seeing more

tangible progress one week to the next.

There are many things that we can do to keep a team's spirits up. When working alone, it's worth sometimes doing the same things to keep yourself going.

WHEN SELF-COMMUNICATION FAILS

Sometimes the issue arises as the solo equivalent to what on team projects we see manifested as communication problems.

Maybe the upcoming schedule needs more or less detail. Maybe this next chunk of work needs to be broken into smaller steps or different kinds of subtasks. Communication is a process, and it's always open to further refinement or simply switching approach. On a solo project your communication challenge is successfully keeping future you working toward the same objectives thought through by past you.

WHEN GOALS ARE UNREASONABLE

Sometimes the issue comes down to what on a team we'd easily

identify as a lack of training or a skill mismatch.

Demanding from yourself unrealistic results on an unrealistic timeline, stretching too far beyond your present level of experience or abilities isn't motivating or educational, it's instead bound to frustrate, stalling forward progress or bringing hopelessness to the surface.

There's always a temptation to try to be a hero, taking on a monumental task just to prove to the world that you can, but the work can't really impress or excite anyone if driving so hard into it makes you crazy or upset to the point that you can't make progress.

World class weightlifters didn't start by trying to lift the amount they do now. It could've snapped their bones, torn their muscles, and crushed their bodies. The consequences of trying to do the equivalent as a videogame maker are certainly less visceral, since

the risks of overexertion are instead purely psychological, but it's a nasty sort of harm that can drag out stealing years of productivity.

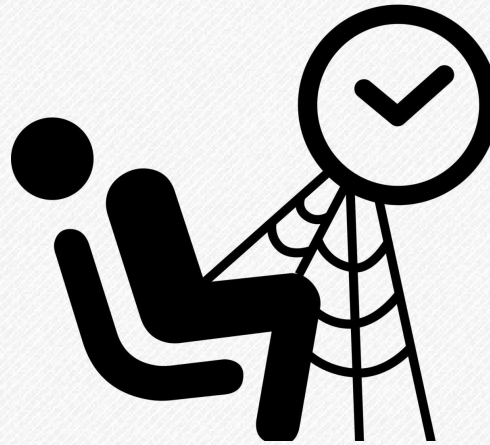
NOBODY'S HOPELESS, NOBODY'S PERFECT

Most likely: you're not 100% off the deep end in any of these dimensions, but you're not completely matured past all of them either. Every person has some mixture of different imperfections in attitude to work on improving.

Even if you're not working with a team, you're still functionally a team of one. Keep an open mind to trying out various ways to make yourself easier (for yourself!) to work with.

DON'T WAIT FOR AN EVENT, JOB, CONTEST, OR ASSIGNMENT

School trains us to do what's asked of us, and most roles in the workplace reward the same. But when you have the power to create from your imagination, don't wait for someone else to ask you to do what you want to do.



As soon as you know how to program in a practical language, are functionally fluent with digital art and audio creation, and have developed one or more videogame projects of your own design, don't wait for an event, job, contest, assignment, invitation, business plan, or context to make what you want to make.

Want to work on an overhead racing game? Close the browser and go make it.

Want to work on a 3D puzzle game, a space exploration game, a scrolling shooter, a sniping mission, a mech combat game, a submarine stealth game, a game where you play as a squirrel gathering nuts or control weather to shape how tiny villages develop? Leave this web site, and go make it.

In case you're still reading:

Set a date for it to be completed by. Whether that's 6 months away or 24 hours away, scale the style,

features, polish, and scope to whatever it takes to make that schedule. Tens of thousands of amateur videogame developers routinely put together playable, original projects over 24-48 hour spans in game jams. There's nothing wrong with game jams – but there's absolutely no reason to wait for a game jam to roll around to create something in 2 days. Or 2 weeks. Or 2 months. There's no reason why it has to only take place a few times a year, while other people are doing it, or in connection to some arbitrary centralized theme.

It may not be clear how to market it, who it's for, how it will turn out, or even why it's being done. Don't let that get in the way. It's probable that at some point in the future, a situation will unexpectedly come up to share and show off something that you've done. Whether or not you'll have something that fits the occasion will depend on whether or not something was put together

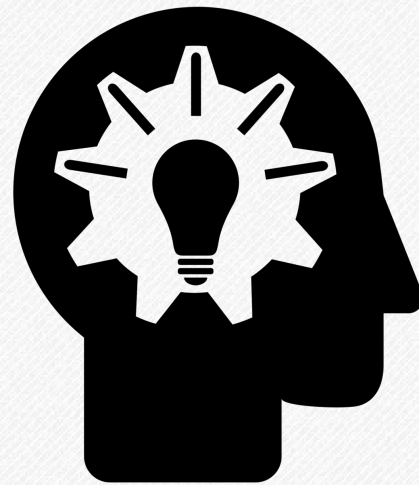
well before there was reason to have done so.

Is there something you want to be making, or want to have made, that you're capable of creating on your own (or creating “enough of” on your own to fill in the blanks for now), that you haven't yet started making?

Then what are you waiting for?

THE BRAIN IS NOT AN EMULATOR

Modern PCs can emulate (play game code from) old game systems. Despite the brain's power, it's unfit for that task. Memories of real-time play and expectations of how an idea will work out are highly undependable.



LessWrong explains verbal overshadowing with the following examples from research:

People who were asked to describe a face after seeing it are worse at recognizing the same face later.

People who are asked to describe a wine after drinking it are worse at recognizing the same wine later.

People who are asked to give reasons for their preferences among a collection of jellies are worse at identifying their own preferences among those jellies.

Or, as the concept relates to this entry: when thoughts about a

“principally non-verbal process” (gameplay) are digested into words, they can interfere with the accuracy of those memories rather than aiding them. The more we discuss videogames and videogame design, the more at risk we become of wandering astray from what we’re supposedly discussing. Talk about Pac-Man too much, and you’re likely to start mixing up all sorts of important details (pop quiz: does Pac-Man move faster or slower than the ghosts? [check your

answer at the end of this section]), if not outright dreaming up and emphasizing all sorts of seemingly important aspects that are either not there or turn out to be comparatively unimportant during real, actual, human play.

Part of what makes this tendency dangerous is that we're unaware of this confusion, since the brain is providing its own reference. Minds are powerful but often imprecise, and we're prone to lie to ourselves on accident. We think we know how something works in a game, so we'll just visualize the game in our heads to check, and... yep, sure enough, it's exactly like we thought. But of course the brain is actually fudging that visualization, constructing it from what we think we know, rather than replaying a lossless video recording.

GAMEPLAY IS NOTHING LIKE WATCHING

Speaking of video recording: looking up a YouTube video isn't sufficient, either, and nor is watching someone else play. The human complexities of analog

input, cognitive overload, attentional bias and reflex limitations are lost in indescribable tacit memory and immediate experience, but none of that happens if you aren't the one actively playing. When you aren't the one playing there's no stress, no relief, no exploration, no dexterity, no practice, no learning, no being lost nor stumped nor excited. Without those events in the picture, it's not really a videogame being discussed at all. Studying YouTube videos of games is a good way to become an expert in making YouTube videos of games, but that's not at all what we're interested in doing.

PLAY A LITTLE GAME

Imagination can't even fully recreate playing classics like Pong or Breakout in real-time. Stop for a minute and try it. Let the actions, movements, and sounds play out.

[Play Pong or Breakout in your head. Seriously. No one else will notice or care. And if it turns out you're good at it, you'll have a new

way to entertain yourself in long meetings.]

Provided that you can at least maintain coherent positions and velocities beyond a few seconds (surprisingly non-trivial), you'll likely need to admit that you pretty quickly started cheating or fudging basic unknowns: messing with movement (analog dials produce a very particular type of on-screen glide!), losing track of brick size, locations, and count (how many per row, and how many rows? brick dimensions? is there a space above the bricks? how large is that space? how many points are bricks in various rows worth?), getting ball bounces wrong (you know the ball doesn't reflect off paddles like it does the walls, right? and that in Breakout the ball never hits the side of a brick, then the ball moves harmlessly through all other bricks on the way down, such that it only hitting one brick per paddle bounce unless it just bounced off the back wall... ?). How big are the paddles? How

fast does the ball move, and when does the ball's speed change? To what score were you playing to in Pong? And to mentally simulate Pong you needed to play both sides – but certainly that yields a very different mental experience than only playing for one side?

Those are some of the simplest videogames ever developed. They're from the 1970's, and built on extremely limited hardware. Meanwhile there are plenty of designers who feel quite certain that they firmly have a grasp in memory on the gameplay from Genesis games, Xbox games, Wii Games, and even PlayStation 3 games – sometimes played years ago, and in all such cases with significantly greater complication in animation, audio, object count, level design, input devices, and other factors than the two comparatively simple games just mentioned. Not only is the memory mostly fudged, but thanks to Verbal Overshadowing the more we try to explain what

we remember, the further we stray from what the gameplay actually consisted of.

DELICACY OF EMPHASIS

The factors I'm referring to may seem like trivial tuning matters, but that's just how wrong the mind gets it: when those aspects are other than they really are (say, in a crummy port or shoddy clone, or in your human imagination) a videogame can take on an entirely different experiential and mechanical quality. Attention and anticipation are fragile but central to gameplay, and get manipulated by countless audio cues, visual clues, input behaviors, and so on. Just one incorrectly remembered element or human response can drown out the real priorities that occur during live play. Super Mario Bros and Super Mario 64, for example, both involve a lot more combat and exploration than designers typically remember when discussing those games, both of which get wrongly abstracted as pure platform

jumpers with satisfying controls. (In SMB, exploration primarily takes the form of crouching on most pipes to check which lead to underground zones, bumping normal-looking bricks to find the ones that are multi-coin blocks, etc.)

There's a temptation to inspect the totality of the system, as though every authored decision has a significant and isolatable affect the player. This simply isn't true. When the brain gets overwhelmed by rich details and possibilities it relies upon performance shortcuts, crude mental models, and selectively ignoring the vast majority of information. An efficient approach to gameplay adaptation won't even internalize most environmental details, but will instead focus on how to react or solve when necessary and how to advance between combat and puzzles. As to what information gets acknowledged or disregarded on the fly for a given videogame, and what the brain actually does

with it, that can only be determined by playing the game (not just having played it at some time before).

Trying to remember gameplay is even more difficult than accurately simulating all the inputs, outputs, and code, because it also requires simulating and accounting for the complexity of the brain. Given that the brain can't faithfully emulate Atari hardware, it certainly can't handle fully emulating itself, too – all while focusing on the player's perspective and experience, of course.

OUR LIMITATIONS

True, we can have a fuzzy memory of some screenshots, a few memorable sounds or fragments of songs, perhaps a single scene playing back as though it were a non-interactive animation (again: minus the stress, confusion, confidence...). We can recall crude facts about our experience, such as whether we had fun, how long we spent playing, and whether we were impressed by it. However

when trying to recreate or discuss a camera movement, player behavior, layout, input dynamic, or other aspect of an existing videogame, there's no substitute for breaking it out and playing it. Then iteratively making notes – not after the experience, while Verbal Overshadowing is rapidly taking over, but during the experience, so as to immediately and frequently check whether the observations and thoughts are holding up.

Often, confirmation bias will still takeover, making things seem to align to the way we think of them, but at least some of the time we can catch our own nonsense before we spread it to others. This seemingly hopeless picture I'm painting is quite deliberate, because the real takeaway here isn't on how to better document gameplay in words to avoid the need to reference gameplay, it's to accept the futility of anything less than actual gameplay to accurately and completely capture what it is.

DESIGNING GAMEPLAY ISN'T VERBAL

The other challenge this introduces is the near-guarantee that most of what we think about when designing and planning gameplay features in our heads (and on paper in words) is pure fantasy that will fall apart and need to be mostly rethought the very moment it's tested in interactive implementation. We can write or talk for arbitrarily long about how we think it will work and how well it will play out, but we can't even reliably articulate that dynamic for rudimentary games that already exist, let alone nebulous ideas in our imaginations. At best, such thoughts hint at possible directions, and give sufficiently clear direction to begin exploration in a tangible way, but quite often once we get the ideas playable we discover that they work or flop for reasons totally unrelated to what we expected.

Because of this effect, often what makes great videogame designers isn't writing better design docs, or

even having better ideas, so much as it's having a better process for building, trying, rejecting, and salvaging ideas, then seeing through the polished completion of those thoughts that survived and fit well together. It's an elaborate dance to compensate for the inability of our brains to know or manipulate gameplay accurately as thought or words alone.



POP QUIZ SOLUTION

So, does Pac-Man move faster or slower than the ghosts? As [the Pac-Man Dossier](#) summarizes:

The game starts with Pac-Man at 80% of his maximum speed. By the fifth level, Pac-Man is moving at full speed and will continue to do so until the 21st level. At that point, he slows back down to 90% and holds this speed for the remainder of the game. Every time Pac-Man eats a regular dot, he stops moving for one frame (1/60th of a second), slowing his progress by roughly ten percent—just enough for a

following ghost to overtake him. Eating an energizer dot causes Pac-Man to stop moving for three frames. The normal speed maintained by the ghosts is a little slower than Pac-Man's until the 21st level when they start moving faster than he does. If a ghost enters a side tunnel, however, its speed is cut nearly in half. When frightened, ghosts move at a much slower rate of speed than normal and, for levels one through four, Pac-Man also speeds up. The table below summarizes the speed data for both Pac-Man and the ghosts, per level.

In other words: it depends on which level you're on (generally, Pac-Man is faster than the ghosts before level 21, except...), if Pac-Man is in a tunnel still filled with energizer dots (in which case the missed movement frames from eating a lot a following ghost catch up), and where the ghost is on the map (they slow down significantly in the wrap tunnels even in the later stages).

What are their relative speeds?
Also covered in that source, if we let X = Pac-Man's speed in levels 5-20, then:

For level 1: Pac-Man moves at 0.8X, Ghosts at 0.75X

For levels 2-4: Pac-Man moves at 0.9X, Ghosts at 0.85X

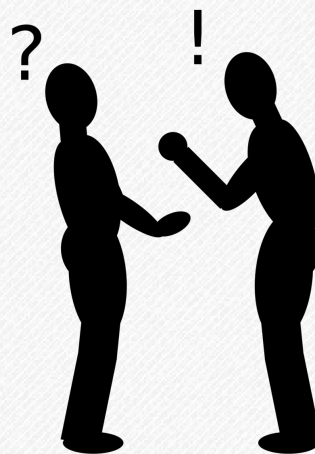
For levels 5-20: Pac-Man moves at 1.0X, Ghosts at 0.95X

For levels 21+: Pac-Man moves at 0.9X, Ghosts at 0.95X

My point, of course, is that the difference is subtle enough in either direction, and complex enough in its involved factors (many of which casual Pac-Man players may not be aware of), that unless you've made a point of learning these facts about Pac-Man through deliberate study, merely being familiar with this iconic classic and having played it on numerous occasions is actually inadequate for being able to figure out something relatively central to how the game plays.

STOP ARGUING ABOUT WHAT MAKES A BETTER GAME

Lots of people have ideas about how videogames ought to be. At the end of the day, the only people who are really deciding how videogames will be are the people that make them. Bring your argument to life.



Stop arguing about what makes a better game, and go make your better game.

Every indie developer that I know – and I’ve had the good fortune to meet quite a few of these folks – has a wildly different answer to the question, “What makes a good videogame?” They generally have an answer, it’s not as though the question leaves developers with nothing to say. My point is that actually saying it doesn’t do much good. The argument usually sounds quite a lot like the game

that they last built or are currently building. As it should.

In purely technical challenges, the questions we deal with are often convergent. In other words, in terms of algorithm or implementation, there are answerable questions about which pathfinding or collision detection routine makes the most sense for a particular application, or at least the pros and cons can be objectively outlined and weighed.

Technical matters exist in other art forms, as well. A painter's opinion isn't what makes a particular material of canvas or paint last, an architect's opinion isn't what determines a structure's ability to withstand an earthquake, and a musician's opinion can't override room acoustics or material realities in instrument construction.

However, in matters of taste, one painter arguing that other painters ought to mimic her own style and purpose would result in a less interesting gallery for the rest of us. Architects and musicians likewise would deprive the world of much needed variety if they spent their energy trying to tell others to do exactly as they personally would do, especially if this were done instead of actually doing it themselves. They could certainly still be aware of one another, be inspired by one another's work, even borrow and remix parts.

Though perhaps the best reason to be acquainted with one another's work is to avoid wasting precious time simply recreating someone else's work, while mistakenly thinking that doing so is trailblazing. We need to know what else is out there not just to learn from it, but also to recognize and reduce redundancy, to not misuse our lives exploring beaten paths. (For people new to making videogames, cloning can be helpful as a form of detailed study or to get initial momentum, but that's neither who nor what's being addressed here.)

The only meaningful yield of the individual's deeply-felt arguments about game design is the actual game designed.

Some people will like it, some people will hate it, some people may feel like it's the game they've dreamed about but didn't know how to make, and the overwhelming majority of the population (even if your game is *Zelda*, *Angry Birds*, or *FarmVille*)

flat out won't really care one way or the other. It's a matter of taste. Even the most popular or critically acclaimed work in any medium is met with infinitely more indifference than with appreciation, or at best in the absolute peak of pop cultural products, iconic recognition without real understanding.

If you're doing a commercial project, sure, you've got to worry about what's going to sell, and that's a different battle than what I'm talking about here. If you're a hobbyist, I still say embrace the hobbyiness, do something strange that beginners can't and professionals won't.

Someone that doesn't like driving games still won't care about the "best" driving game ever created, by any metric or definition of best. The same is true for puzzle games, shooter games, platformer games, and though I suppose it goes without saying, the whole vast and infinite sea of games without clear genres which suffer,

unlike their cousins in clear categories, on account of having a far harder time being found by people that can know before playing that they want to play what it is. You could make exactly the game you mean to make, totally love the outcome, and the person you would've otherwise argued about it with still may not be into it at all.

Of course, since the brain is not an emulator, arguments about what makes games better are even more silly. In argument form, as opposed to videogame form, there's too much information left unspecified. Even if it were all somehow conveyed exactly, the mind would derail managing it all within the first second of execution. In my mind I can fool myself into feeling like any videogame concept is appealing, equally well regardless of whether in reality it's my least or favorite game. If brains could be trusted with juggling all the details in real-time, and if language could do

justice to explaining those details, we wouldn't be throwing so much time into programming and creating assets.

Make the game your argument leads to, to get the argument out of your system, so that you can come up with another argument, informed or changed somehow by how the last project came out. Then make a game out of that new argument, and repeat. It's barely meaningful to ask whether an argument about taste (not an estimation of "the most popular taste" or "fitting the taste of vocal critics" – but actual, individual taste) is right or wrong, but you can be sure that nothing will make the argument more clearly and loudly, in a way unmistakable to those within or interested in our domain, than actually making a videogame out of it.

This entry was inspired by the quote "Complain about the way other people make software by making software," from Andre Torrez, which I found in Steal Like an Artist by Austin Kleon

START BEFORE YOU HAVE AN IDEA

Too often, people get stuck trying to come up with the perfect idea before they'll let themselves start. Then, unsurprisingly, the idea has no connection to what they are able to create! Let the idea evolve from the work.



I received this message the other day:

“I don’t know how to come up with an idea for a game. My problem is that I can’t come up with an original idea. I had many ideas for games, but it turns out those have already been created by other people.”

As always, I really appreciate hearing questions from readers and subscribers. It helps ensure that I’m writing content addressing real questions that people out there are currently running into.

This is a common source of paralysis when trying to get started. Let’s look at some ways of bypass these barriers and get back to game making.

YOU DON’T NEED THE IDEA TO START

Contrary to popular belief, it often works out pretty well to come up with the idea after starting it. This is not the chicken and egg problem that it appears to be.

Start building something. Pretty much anything! Anything simple enough that you realistically could finish in a straightforward,

predictable way. Make “just a racing game,” “just a puzzle game,” “just a strategy game,” “just a platforming game,” etc.

Ask yourself what’s the bare minimum basics needed to be coherent as one of those recognizable game types.

Just start piecing it together.

The different ideas, including what’s going to make it special, will often develop midway from the details, process, and personal interests, strengths, and constraints.

HOW THE IDEAS EMERGE

While dealing with those parts and watching it take shape in stages, you’ll begin bumping into all kinds of ideas of different things to try. Different ways to approach common problems. Different directions to take the game in. Different things to let the players do.

Some ideas arise from incomplete implementation, a discovery in the cracks of work that you can latch

onto and grow into something deliberate. Others may take form while trying to figure out a simpler or more time efficient way to accomplish some immediate goal, involving some approximations or accepting certain limitations. Ideas can also take shape as just needing to fill in for something in a way that you’re able to do well (or well enough!).

Doing this on the fly helps find the workable intersection between current capabilities and current curiosity. Often the key “idea” of the game that really sets it apart isn’t even something easily described at first, but is instead something subtle in the gameplay that only arises from back and forth tinkering with the machine.

WHAT TO DO

Step One: if you’re not already actively working on a project, pick an old or otherwise relatively simple type of game or two and start trying to implementing parts of it to get some momentum. Get yourself further along in the

process. Create a situation in which you can have ideas that you'll be able to promptly put into action, in a context that's at least partly functioning.

Only with the foundation of some code and functionality in place can the imagination begin to orient itself in concrete possibilities rather than random dreaming.

Plenty of concepts that are interesting to think about might not work particularly well in a videogame, or might need a huge budget and large team to get it done. Using this approach you'll always be building in a way that necessarily fits in a game and – just as importantly – within (or just at the edge of) your present skills.

You're not someone else. You don't know the same things, care about the same things, or work the same way. Your work simply isn't going to wind up the same, unless as a deliberate practice and learning exercise you go out of your way to specifically copy

some particular example in as much detail as possible.

HOW WELL IT'S DONE MATTERS MORE

Concern over originality at the idea level, rather than in the details and execution, stems from overemphasizing the importance of the idea. To put it bluntly that's just not really how media, whether entertainment or art, works. Of course the topic matters to an extent. It can drum up certain kinds of excitement or imply connection to certain audiences. However, much of its impact resides in how well it's done in the eyes of the audience that it reaches.

Star Trek didn't invent space ships and science fiction. *Quake* didn't invent zombies, grenade launchers, or gothic(-ish) architecture. *Godfather* didn't invent mob films. *Super Mario Bros.* didn't invent platforming or saving captured princesses. *Back to the Future* didn't invent time travel. These are so well done that

they captured people's imaginations.

This isn't just about TV shows, games, or movies. Moby Dick didn't become a classic because the idea of whale hunting is cool. Moby Dick is so well put together that it creates interest in what it's about.

The idea of Mona Lisa is... well, you get the idea.

Pick something to do, practice to do it better, and you'll discover your own tricks along the way to make them more personalized and unique.

RAPID PROTOTYPING

The other approach to come up with gameplay ideas that haven't yet become a common pattern is to rapid prototype, which requires a high degree of development fluency and often returns a relatively low yield. However, it's one source for original and decent things that aren't evolved from things that began as clones. I took

this approach for InteractionArtist.com

That's still thinking by building. It's just not starting from as established a foundation, accepting a higher chance of producing something uninteresting in exchange for a shot at coming out of the process with something more unusual.

DON'T WAIT FOR INSPIRATION

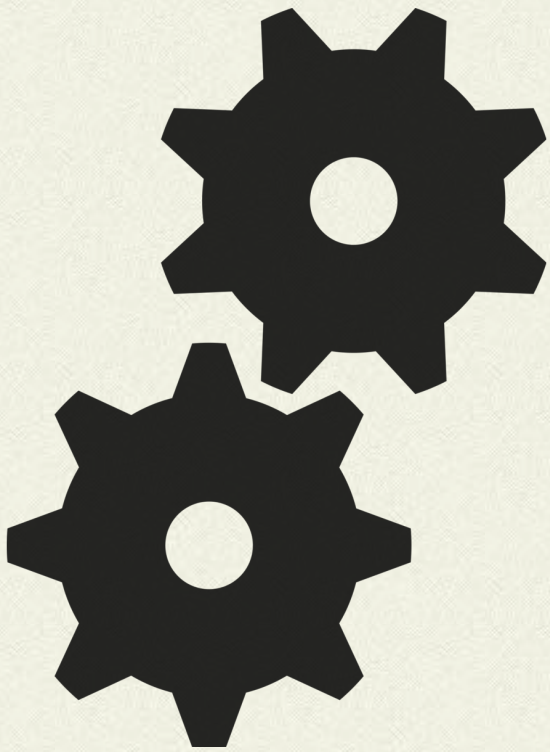
Waiting for an original idea to just happen is an unproductive trap.

Ideas grow out of action, iterative building, and through collaboration with others that have different tastes and strengths than your own.

Want to come up with more ideas? Start making something.

Even if a project starts out as totally unoriginal, with persistence and some practice it'll soon lead to directions you wouldn't have thought up otherwise.

5

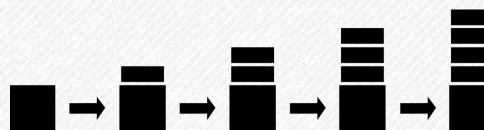


Some game development books begin with game design. But first learning some programming skills as a foundation can help give you the power to put your designs into practice. Let's next dig into some design concepts, since you're now better prepared to try them out.

GAME DESIGN

MODEST FIRST PROJECTS AND INCREMENTAL LEARNING

No one expects their first film project to be just like a Hollywood movie, and yet a similar belief is oddly common among people new to making videogames. Here I make my case for realistic, incremental goals.



Via @HobbyGameDev on Twitter I asked people why they haven't started yet. For the most part the responses helped me better understand challenges new game developers were experiencing. I love these community interactions since they help keep me firmly connected to people's real needs.

Although I was able to field a number of exchanges with just an @reply tweet or two, the most important one to address that couldn't fit in that space is perhaps this exchange:

Hobby Game Dev @HobbyGameDev · Mar 15
People who wish to make videogames but haven't finished your first one yet: what do you feel is the main reason why you haven't done it yet?

@HobbyGameDev · Mar 15
Because all tutorials force me to focus on making Pong or Asteroids first. I want to make MY game and learn as I go

Hobby Game Dev @HobbyGameDev · Mar 16
what is your target game like that you would like to learn about game development through the process of making?

@HobbyGameDev · 23h
Would look-n-play like current (Uncharted, CoD): platform, AI, etc. World building. Plus my secret...

Hobby Game Dev @HobbyGameDev · Mar 16
You want your first game to look and play like games that took 85 full-time professionals 2 years and \$20 million? But more features?

@HobbyGameDev · 19h
Better to fail by trying than by fear. Less discovery needed: parameters known? Create 1-level POC, get funded then iterate.

I want to be clear that I am in absolutely no way intending to draw any negative attention, attitudes, or ill-will toward our good friend in the world, Orange Rectangle (or O.R., for short). I asked my question with the specific purpose in gaining a better understanding of the challenges people were experiencing, and O.R. then offered an answer in good faith to help me. I attempted to offer some real details (those numbers are from Uncharted 2, by the way) in the sincere hope that this might help lead to a startling realization. It did not.

If I thought O.R. was just trolling, I'd simply ignore or block them to prevent any further attention being stolen from the many people out there who are really interested in making videogames. However I've met and spoken with enough people face-to-face sharing a mindset similar to that of O.R. to

believe this is worth addressing directly. That this is still a fairly common way of thinking for people getting started out seems to me a sign that it hasn't been suitably explained, or at least not yet in a place that's easily found by those who are looking.

I offer the following to our friend O.R., and of course to others who may share his mindset:



Dear O.R.,

I'm not saying you can't or shouldn't. Maybe you'll pull it off. That'd be awesome. More power to you.

I suspect many people doubted Notch when he started work on Minecraft. Although by that time he had already been programming for 25 years. People were probably skeptical of the team that made Angry Birds. That may have just been extrapolating from the

51 games that Rovio made before that project became a new standard for mobile gaming. The success of Super Meat Boy was not guaranteed. However Tommy Refenes had been making games for 18 years before that, and Edmund McMillen, Tommy's collaborator on the game, worked on 14 finished games before Super Meat Boy (including its free Flash precursor, Meat Boy).

Even with their accumulated experience, these now famous developers still didn't make games with the look and feel of a modern Call of Duty or Uncharted. From a technical, team size, and content creation standpoint, Minecraft, Angry Birds, and Super Meat Boy are tremendously less complicated than either a Call of Duty or Uncharted sequel (let alone a brand new intellectual property).

Many videogame

developers at some point bite off more than they're ready to chew, for at least one project, and can sometimes take away from that experience some helpful concepts for later. For me that was Guinea Pig back in 2004.

So, with that in mind, why not just attempt the dream game first, and if it doesn't pan out, just learn some lessons in the process?

Here are a few reasons that I think should at least be considered.



A screenshot from Guinea Pig, for which my engine used skeletal animation (authored by a custom tool), clothing and skinning, blood/bullet decals, real-time fully destroyable terrain, dozens of weapons types, parallax, dynamic fire, hi-res "mode 7" semi-3D effects...

WHY NOT START WITH UNCHARTED/COD

I'd like to make a case here for why those tutorials don't start with a Call of Duty or Uncharted style game (let alone a variation with world building and other additional features), and instead focus on games like Pong or Asteroids.

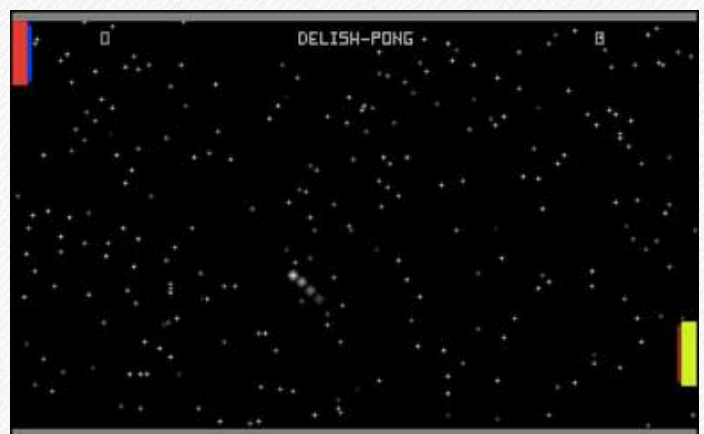
In Daniel Coyle's *The Talent Code: Greatness isn't Born* he talks a lot about deep practice, or edge practice. As a takeaway from his multi-year international study of what world class experts in a variety of disciplines consistently do to get there, he explains:

"...you can capture failure and turn it into skill. The trick is to choose a goal just beyond your present abilities; to target the struggle. Thrashing blindly doesn't help. Reaching does."

Or, if I can reframe that in terms of a more widely recognized system of incremental learning: no matter how strong, flexible, and quick someone is, there are still steps that they need to progress through in order to earn a black belt in karate. Skipping over essential

steps means instead of productive edge learning atop a solid foundation, a person instead tends to wind up with sloppy, ineffective actions detached from the many lessons figured out by prior generations.

While I do not claim to be a world class expert, as Coyle's subjects are, I have enjoyed some recognition for my commercial videogame work, which started more than a decade prior by doing exactly the same kinds of simple historical game remakes that I consistently encourage others to try when starting out. By starting



This Pong remake was some of my first experience with AI. Around the time I was playing (and modding a bit for) Quake 2. I wasn't crazy for retro games – I didn't even play Atari's original Pong until 8 years later – but I was interested in creating games that I could finish and strive to make well.

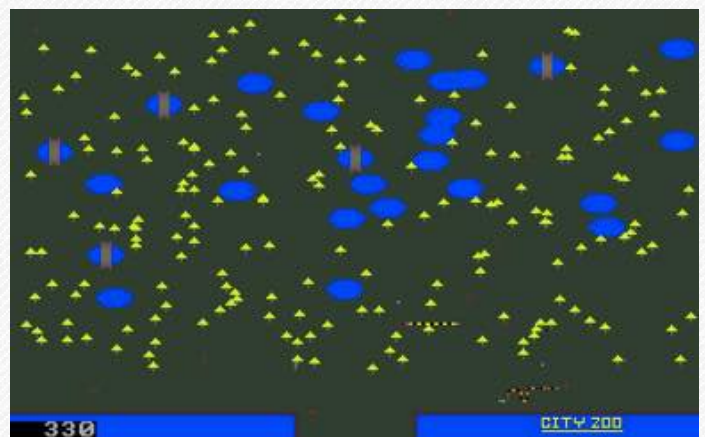
small and moving toward increasingly complex games, a solid foundation of applied design principles and practical coding habits can be formed incrementally, with lessons picked up along the way that are appropriate in size to be learned from.

Over the years I've now seen a number of people massively overshoot their means on an early project and fail spectacularly. In most cases that unfinished game is the last one they work on, never fully recovering from having dug a pit of embarrassment and fighting through considerable frustration every step of the way. Getting some practice before diving headfirst into these kinds of undertakings can help steel us to the complications that they entail.

For example, by the time I bit off more than I could chew on Guinea Pig, I had already completed more than 15 other smaller games in the years prior. That steady momentum beforehand gave me

the ability to pick myself back up and promptly get back up to full speed on new projects when that one didn't pan out.

This progression of foundational skills was also a central philosophy to both videogame development clubs I helped start: before someone has been a lead or solo developer for a game of early 1980's-gameplay complexity, they're typically not ready to lead a team of others on a game of late-1980's gameplay complexity, and so on. This begins with late 1970's scale: Snake, Breakout (not Arkanoid, that's mid-80's!), Asteroids, or Space Invaders.



My first graphical game was based on the late 1970's game Snake. Again: not because I grew up with it – I was born in 1984 – but because in terms of game mechanics and programming it was a realistic start.

For those developers that truly are extraordinarily efficient and clever, who say “that’s so easy, why bother?” I have two responses: (A) that’s grand, then you’ll have no problem knocking it out in 30 minutes, or in an evening at most, after which when you come back with it completed you’ll have impressed and earned the trust of more potential collaborators. And: (B) even if you know how to get started, and know that you know enough to work your way through it, going through the actual process of doing so will involve sorting out some details, processes, unexpected complications, and chunks of code that will together speed up and simplify working on the games you take on later.

Being able to do something and having actually done it are not the same thing. In particular, after you’ve already done it you’ll have expanded the domain of what you’re ready and able to do next.

BEING UNPREPARED ISN’T BRAVERY

As my friend Nicholas Brown commented:

“Shooting high and learning from it is nice but at a certain point you’re trying to shoot down an airplane with a bow and arrow. It’s just not going to happen and you probably won’t get anything useful out of the experience. Saying ‘that project is out of my scope/skillset’ isn’t fear, it’s good sense as a developer.”

There is a certain amount of fearlessness, unreasonableness, and boldness that successful entrepreneurs, innovators, and cultural leaders of all kinds have had at their core. However, the fact that some amount of fearlessness is a factor in success does not automatically mean that an even greater amount of fearlessness linearly correlates to greater success, nor that it’s the only factor. Bold leaders still need a viable strategy, considerable experience and connections in the relevant domains, and a certain dose of patience and discipline underlying their persistence and determination.

At the heart of Coyle's research in *The Talent Code* mentioned earlier is the idea that we can learn most from those experiences which are just beyond our reach. That's at the core of deep, edge practice. When we extend ourselves greatly beyond our past and present proven capabilities, there are often too many convoluted ways to make mistakes for us to figure out how to make sense out of what isn't going right, let alone to learn from it.

FIRST PROJECTS ARE ALWAYS ROUGH

We tend to learn most rapidly when we're still very new at something. Anything about the field can be absorbed as new learning. We quickly overcome the awkwardness and inefficiencies in dealing with whatever development environments, programming languages, and content tools we're working with. If you put a few months into learning how to paint landscapes well, your work at the end of that time would likely bear no

resemblance or fit to the first ones attempted. Because of that rapid learning, going in this case from one new painting attempt to another gives practice and opportunity to rethink the initial decisions, and will likely work out much better than just spending all of those months retouching the very first painting started.

As Coyle (again, from *The Talent Code*) explains about the famously talented and accomplished Brontë sisters in relation to their work as novelists:

"...the myth Barker upends most completely is the assertion that the Brontës were natural-born novelists. The first little books weren't just amateurish — a given, since their authors were so young—they lacked any signs of incipient genius. Far from original creations, they were bald imitations of magazine articles and books of the day, in which the three sisters and their brother Branwell copied themes of exotic adventure and melodramatic romance, mimicking the voices of famous authors and cribbing characters wholesale."

Because our first work is inevitably rife with beginner errors from learning as we go – even world famous authors first wrote amateur junk – insisting on making an idealized dream project as your first undertaking is unfair to the vision, condemning it to be rife with beginner errors. This is instead an ideal time to experiment a bit with modeling the proven past successes of others (those that are reasonably achievable within our means), including simple historical cloning as a form of initial practice and not as a shady business scheme. The Brontës started that way.

I'd say try getting some of the junk of your system first on titles that can be finished far sooner, aren't as personally important (read: not yet highly original), and won't involve pulling others into the risk involved (even if not risking your and their money, then at least out of care and respect for their time in something that has a

reasonable chance of not finishing or finishing well).

COMPOSING WITHOUT PLAYING FIRST

When someone wants to play piano, do they begin by immediately composing original works? I don't doubt at all that in the history of all civilization there are people who have tried to do that. Generally, though, we have not heard of them or their work.

We start with learning a bit about notes, sheet music, and scales. We build up to childishly simple sequences like Mary Had a Little Lamb. We move up, through consistent and focused practice, to trying to merely perform well some the more advanced classical works far before trying to compose our own.

The reason we see so many introductory materials about games like Pong and Asteroids – including the Hands On Intro to Game Programming (my own approach to that path) – is because those are the Mary Had a Little Lamb of videogame creation,

a tried-and-true way for someone new to game development to get oriented and learn to perform full, simpler classics before trying to become the next Bach or Mozart from day one.

BUILDING AN AIRCRAFT CARRIER... FIRST

Where the above music analogy breaks down is that composers are often able to write music primarily alone. While that is true for a subset of videogames, it is not the case for the size and scope of videogames that you've identified as your target.

If someone is interested in making a boat and decides that the kind of boat they wish to design is an aircraft carrier, that's necessarily going to require a massive scale of teamwork, money, experience, earned trust/credentials, compromises, and a lifetime of work leading up to that point. It either won't be the first ship they work on, or the rushed result will involve so many compromises that it will not come anywhere close to

what is typically expected of this ship type.

While it's true that we don't have many of the same material constraints and labor challenges as ship builders, what we are unavoidably faced with are literally millions of design, artistic, and technical decisions, from the very large to very small, which we have to make about everything in the game.

Faced with all of those decisions, making them well at every step either means having the experience from enough variety of past projects to make informed tradeoffs between alternatives (which is why teams generally want professionals experienced in videogame making, not just people who "can program" or "know Photoshop"), prototyping like crazy to find even a few small innovations that work, or adhering fairly strictly to conventions.

However, note that even companies filled with experienced,

full-time professionals creating massive games that do adhere to most conventions of their genre still manage to go through \$20 million just to get their game done and complete. These companies are in the business of maximizing profits and minimizing costs, with both internal and external pressures to figure out ways to do more and better work with the lowest possible expenses.

It still costs them \$20 million or more to make what may appear to be a formulaic and relatively straightforward followup to an existing franchise (note that the actual changes and additions involved are often quite complex, but the differences may be difficult for someone who is not on the team to fully appreciate).

AN IDEA IS ONLY AS GOOD AS ITS EXECUTION

One of the other common themes that arise in advice from experienced developers for others just starting out is along the lines of “ideas are a dime a dozen” or “no one cares about your great

game idea.” This is in regard to what you’re calling your “secret” that you suggest will, alongside world building, set your first game apart from Call of Duty and Uncharted.

I often introduce this concept to others with an example like this:

“I’m going to make the world’s best painting. I have an idea way better than the Mona Lisa. That one’s just a picture of some girl.”

Most successful games aren’t a particularly genius idea. Conceptually they’re usually pretty blatantly derivative but are polished in their execution to an amazing degree. When the idea is at least unusual, perhaps one that has been done before but not well enough yet to achieve widespread commercial significance and recognition, at best that concept serves as a marketing point to get people talking about it. But people generally don’t keep playing a game, nor recommend a game to their friends, based primarily on whether a game has an interesting

idea, unless it also has world-class execution and refinement.

Provided it's got world-class execution and refinement, an interesting or original idea is often not even needed.

PICK A FIGHT YOU CAN WIN

Sometimes there's a confusion that because one or two indie games succeed financially in a genre previously thought dead (point and click adventure, for example), or with a very weird type of game (much of what's found in IndieCade or IGF), those games have been proven to be economically viable after all.

The underlying misunderstanding though is that what may be a tremendous amount of money when split only 1-8 directions among a small team of indie developers working out of their homes or a tiny shared office does not even show up on the radar of the kind of money that AAA companies burn through in only a few months of salaries for massive teams comprised entirely of

experienced developers, legal/business personnel, and support staff along with paying for sprawling office complexes, retail distribution arrangements, television/billboard advertising budgets, etc.

Part of the way this all holds together is that even though a few individuals (out of the many, many more trying) might make enough money to pay their own costs and then some, it's still often far from being something that the big studios can take seriously as a business opportunity.

In 2010 I made a strange iPad entertainment app that paid my rent for 18 months. That registers as a medium success – weirdly putting me somewhere in the upper percentile in terms of return. While it clearly didn't make me wealthy (my rent at the time wasn't much), many more developers lose considerable sums of money or generate very tiny revenues in the range of fewer than dozens of dollars from ads, virtually non-

existent sales, etc. Also, as a fair warning: the App Store was definitely a lot less of an over-saturated mess in 2010 than it is now. (Speaking of which, back in 2008 I developed a game for the app store that earned considerably more than that... for the publisher, which is how I learned a lot about being the weaker party financially in a negotiation.)

Anyhow, for perspective, a massive publisher pays their CEO more money than that 2010 app's total earnings in what calculates to about 24 hours of their time. That's not including the salaries of 9,000 other people, the costs associated with massive facilities, or the remaining mountains of money expected leftover for annual profits, etc. They have truly staggering ongoing costs to keep up with, which means they can't afford to take risky chances on little weird stuff, or on genres that are no longer desired by mainstream players. They deliver

on what they have solid evidence to think will make a sizable return on their investment. Projects that don't meet that goal get eliminated. They're a business.

This is why when we see advanced indie developers succeeding commercially, it's usually not by doing better at what the big companies are in position to do well, but instead from some combination of strange projects (with regard to their execution, often not in the main idea; i.e. tons of personality in an otherwise relatively simple experience), retro genres that the AAA companies haven't been able to justify for 10+ years due to the shrinking level of market demand for such games, or the intersection of both (meaning tons of personality in a retro genre that has otherwise lost most of its market significance). With the rare exception of some mod teams that quickly get swallowed up into the machine and vanish for years doing polish work, indies stick to guerrilla

tactics in niche domains that the huge companies can't justify meddling in.

It's not a matter of purely creative vision that Minecraft is built out of 1-meter cubes textured with pixel art, instead of having the look and feel of Uncharted or Call of Duty. It's small team resourcefulness.



Minecraft is basically 3D pixel art.

FUNDING WILL AFFECT DESIGN

Getting funded does not mean that you have the freedom to design your perfect game. When money gets involved, it tends to influence what gets made, even when sometimes only out of some crude evolutionary feedback loop where the games that do well in a given market space tend to soon

be followed by variations from competitors.

But what about something like Kickstarter, in which customers prove that they're there before development even begins? The space is filled with people that, because they've never had to work with a real budget before, tend to not know how to schedule and plan a commercial project.

If Kickstarter sounds like freedom, it's worth looking up Code Hero, which raised \$170k, well past their \$100k target. I'll spare you the web search, and share the major update from last year: effectively, they ran out of money long ago and are now an entirely volunteer effort. They appear trapped with it.

WHERE I'M COMING FROM

Although I started as a hobbyist, returned to being a hobbyist, and help other people get into videogame making as hobbyists, what I'm suggesting about the nature of huge commercial-scale videogames is not just guesswork.

I've worked professionally as a game designer on the size of games and teams that you're talking about: games that take years, 80-200 full-time professionals, and tens of millions of dollars. I've also worked professionally as a game designer at other commercial scales: at a casual games start-up, in a gameplay R&D company, and as an iPhone game developer both independently and through publishers. I am familiar with at least a cross-section of commercial videogame development.

More importantly to this exchange I have also, with zero budget, served as a lead or solo developer on more than 85 completed freeware games with 3-18 month production schedules, and hundreds of same-day playable prototypes. Some of those were for class projects, some were created in clubs that I established, and many were just developed on the side.

Being in a position to compare those professional and nonprofessional experiences, I can say with total certainty that I have found infinitely more creative freedom in the projects that have no budget or funding than I've seen in working on games with a big budget.

That applies no matter whether I was being paid steady salary, hiring/paying/leading a team of others, funded by outside investment, or even when completely self-funding projects and theoretically having total directorial control. Games that cost money to develop have a certain obligation to prioritize earning (at least) that money back, which places certain inflexible outside demands on the work.

MAKE FINISHED GAMES. WITHOUT FUNDING.

There are ways to make videogames that don't require getting anyone else's approval, taking anyone else's money, or handing over veto control to your design ideas.

It takes time and a lot of work.
There's no easy shortcut to it. But
it is a way.

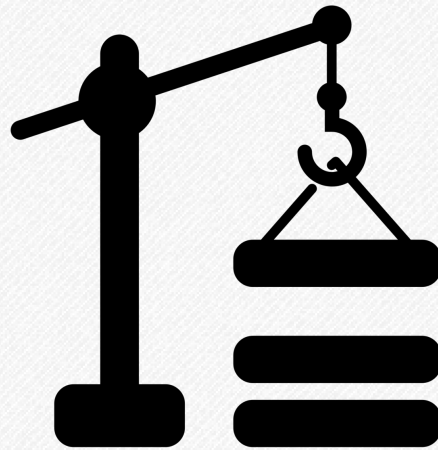
That way is to start small, building
your way up from modest
beginning principles based in
time-tested classic designs,
toward the point where you're
prepared to do your more grand
and elaborate game ideas justice,
so as to have the ability to realize
them without making
compromises in service of
financial obligations.

In other words, the way is to make
videogames for free for awhile,
incrementally advancing your skills
and design understanding with
each new project.

If however you choose to still push
on the route of seeking substantial
external funding, especially
without any track record yet of
completing any simpler games, I
wish you only the best. Good luck.

BOTTOM-UP VS TOP-DOWN GAME DESIGN

There are infinite ways to do and discuss design work. A key concept to ground us is that there's a continuum between how much your game comes from a central, planed idea as opposed to growing out from its parts.



There are a few things that I would like to say about bottom-up design, compared to top-down design. Before I can do so, I want to make sure that we're on the same page as to how I use those terms – they mean somewhat different things to different fields, and I suspect it's likely that some readers have never yet come across these handy terms for discussion.

Top-down design starts from what a mind wants, and shapes that into something a medium can do.

Bottom-up design starts from what a medium can do, and shapes that into something a mind wants.

For example, a top-down art process might begin by thinking, “I want to create an image of Napoleon, in a way that conveys great intensity – this concept is worth attempting and sharing.” The artist would then create the most complete or faithful realization of that idea possible, using whatever tools and format the artist is most proficient in



Napoleon crosses the Alps by Jacques-Louis David
Emotive representation of an existing concept of interest
Examples: Call of Duty Modern Warfare, Madden



Impression, Sunrise by Claude Monet
Concept inspired by or chosen partly by its form of representation
Examples: Mario Bros, Pac-Man, Asteroids

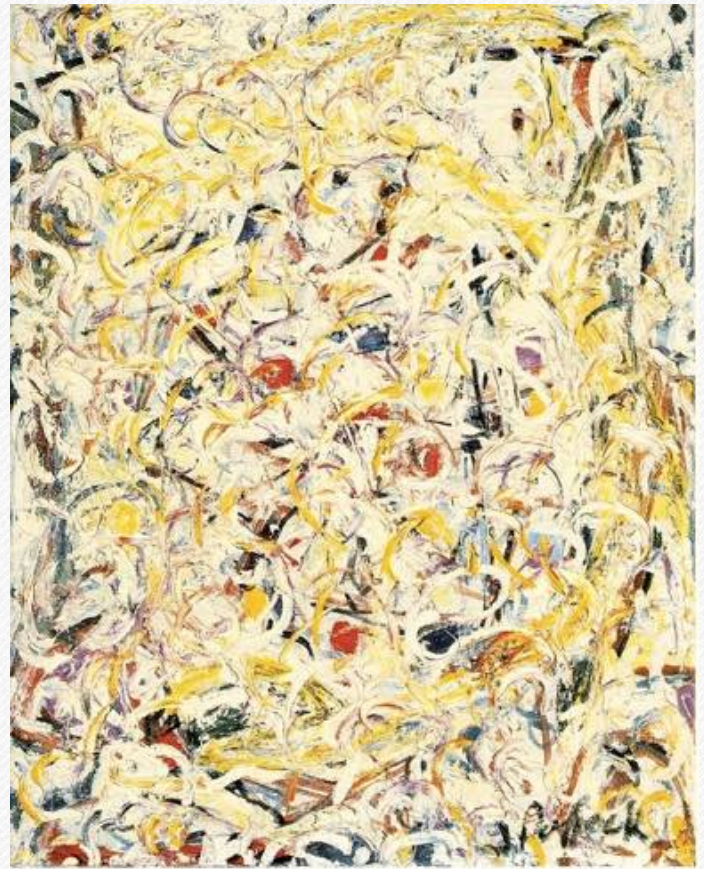
particular effect – I will find a subject that emphasizes the beauty of this effect.” The artist would then seek a technique and subject particularly well suited to make the most of what effect the artist’s tool and format convey. The resulting work is as much or more about the paint and canvas itself as it is about the subject matter; the grain and texture take part in the creative process, somewhat like a sculptor carving stone in a way that not only considers existing fracture lines, but perhaps even calls attention to them.

(might be charcoal, oil paint, or 3D Studio Max). The goal is to realize an idea, and if it can be made less evident that this is done by oil paint applied to canvas, by making its focal point what it is showing instead of method by which it is shown, that is desirable in this case.

A bottom-up art process, by contrast, might begin by thinking, “The texture and shine of oil paint on canvas produce a very

Naturally, there is a continuum, or smooth gradient, between these ideals. It is not as simple as being one or the other. By virtue of being represented through a medium, the work will necessarily account for constraints and affordances of how it is made; likewise, by being the work of a human being, it inevitably will come into contact with how the human mind conceives of reality. There is a significant difference, however, in whether the work starts mostly as a chosen idea or mostly through exploratory execution, and that difference is generally evident in the end result.

As a rough test: if a description of the project sounds normal to someone that doesn't play videogames (or, in our ongoing analogy, study art), then it is probably top-down. The more eccentric, absurd, and arbitrary the description of what the player does (or what a painting is "about"), the further it probably is



Shimmering Substance by Jackson Pollock
It is itself, not a representation of some other idea
Examples: Qix, Peggle, Bejeweled, Tetris

toward the bottom-up side of the spectrum.

For example, Bubble Bobble is a game where two dragons trap toys, witches, and drunks in bubbles to pop them then eat the fruits, vegetables, and candies that fly out, though if they take too long then the ghost monster chases them until they either die or kill the last enemy – that's bottom-up, because the game

mechanics clearly were not invented to support that haphazard concept, but the other way around. The same is true for Super Mario Bros, Joust, or Breakout. It's not primarily about wish fulfillment, like being a superhero that fights crime or a soldier that wins the war for freedom – it's first and foremost about how what's going on is going on, the what being secondary and subservient to the how. The texture of the “paint” (software) and “canvas” (hardware) are central to what these videogames are, and celebrated instead of hidden.

The continuum extends farther off in both directions, too. Miles below my previous example of bottom-up, but in that same direction, an artist might think, “The texture and shine of oil paint produce a very particular effect – I will explore this effect,” thus abandoning subject altogether.

When the Wii came out, developers began asking, “What

sort of interesting things can we do with this Wii Remote?” That was bottom-up design. When people began thinking in terms of designing games for iPhone that used multi-touch, gravity data, saved photos, or GPS, that was also bottom-up. The first racing games were made partly because racing is cool, but mostly because that was one of the few things videogame machines at the time could represent. Likewise for space-based shooting action – an all black background with no buildings or flora to collide with was the sort of thing primitive devices could represent.

That Pong is a crude representation of “tennis” instead of football or baseball was not a matter of its designer being a tennis player, nor was it being designed to satisfy tennis fans.

Racing the Beam by Ian Bogost and Nick Montfort provides a brief history of how the Atari 2600 hardware limitations influenced (and in many cases inspired) the

games designed on the platform. Yars' Revenge was co-designed by Howard Scott Warshaw and what-an-Atari-2600-can-do.

By contrast, when the PS3 came out, developers felt less constrained than ever in terms of computation, and began dreaming about how they might share new stories (Heavy Rain, Metal Gear Solid 4), fantasy characters and settings (Batman Arkane Asylum, Final Fantasy sequels), and appealing concept pitches (make your own physics game in Little Big Planet). That's top-down.

When a modern videogame is about tennis, racing, or space combat, it is no longer because that's one of the few physical events that a computer can process and display.

Even though many modern games now have the luxurious option of more faithfully representing top-down concepts, and even though bottom-up has largely taken place at the dawn of platforms utilizing

innovative controls and graphics (vector to raster was as huge a shift as 2D to 3D), there is nothing ancient, unsuitable, or undesirable about bottom-up design. Part of why I opted to show the paintings is that the Napoleon is from 1800, Impression is from 1872, and Shimmering is from 1946 – that is, it took a lot of time, genius, and creativity for humankind to reach the sort of bottom-up, format-inspired thinking that the earliest videogames employed.

An inexperienced game designer, yet unfamiliar with the grain of the machine – what impressive things it can very easily do well, what somewhat less impressive things it can hardly handle, what the user's relationship is like to the experience through the set of controls provided – most often thinks in terms of top-down design. We're all familiar enough with films and storytelling to think about what happens, where, and to whom in a story, but the typical level of procedural synthesis is

that which only knows what it likes when it has tried it, and will thus “borrow” a mountain of assumptions about gameplay and interaction from another game. A certain sort of confidence and understanding of the technology available is necessary for bottom-up design to even be an option.

On this note, to build a clone or derivative of another game, either completely or in terms of core gameplay, is top-down. The elevator pitch for Goldeneye 007 on Nintendo 64 fits in this category. To paraphrase: “It’s like Virtua Cop – enemies react to where they are shot, and the gun can move freely about the screen – except not on rails, and it takes place in the James Bond universe with the player completing missions as James Bond.” Making a videogame version of Monopoly or soccer is top-down, of course, but so is making a game “like Super Mario Bros” or “like Tetris“, even if the original game’s roots were bottom-up design – no less



*Façade by Procedural Arts
It gets awwwkkwwaarrddd.*

than “resembling Napoleon” is rooted in the complete concept of Napoleon that existed before and outside of Jacques-Louis David’s painting.

An important thing to bear in mind about top-down design is that, despite the remarkable hardware advances of the past several decades, we are still a long, long way from being able to take a purely top-down approach to videogame design. Façade is the closest thing that we have to *The Great Gatsby* – or maybe *The Sims* or *Second Life* are no further, coming towards it from different angles.



Half-Life 2 by Valve. We were quite excited in 2004 when there was 1 room in a game where it mattered that objects have weight. The gravity gun and physics gameplay were bottom-up design within an otherwise top-down world.

Even though PS3 and modern computers give a lot flexibility to think in terms of, “Imagine a world, in the distant future, where...” there’s still a few qualifying sentences implied after that: “In this magical place, all characters will repeat the same things if spoken to multiple times, they’ll have a small set of awkward poses they can switch between to convey expression, and the player will speak by choosing statements from a list of 3-4 pre-written statements. There are only a few things that can be done to interact with the relatively small

percentage of the world that can be interacted with, and the solutions to challenges will either be those that we explicitly planned, or possibly some slight variation by clumsily moving objects around in the game’s exaggerated physics.

Top-down is unlikely to free us from limitations, because by its nature it echoes better than it speaks. It adopts the additional constraints that shaped previous gaming and story conventions much more naturally than it pushes the boundaries to test where they end.

Innovation – particularly the surprising, unintuitive innovations that no one could have seen coming – tend to come from bottom-up. It is no coincidence that the genrification of the industry, the attack of the clones, has coincided with the increase in hardware power inviting more top-down design, while those pockets of gaming that see the most aggressive innovation – indie

games, mobile games, experimental games, and little web games – tend to be bottom-up (at least, those that don't rely heavily on appeal to retro fandom).

The imagination – the source of top-down – copies but personalizes, mixes and matches what it has already experienced, or even when it thinks it is finally being clever and original, has been swept up by the zeitgeist of whatever recent films, best sellers, and news stories have everyone thinking about. There is, of course, a market for this; people like what's recognizable, and people like getting what they're looking for. But as the top-down market is predominantly derivative, from a creative standpoint it's more like being a translator than like being a writer. For the craftsmen and artists among us, it is not our desired line of work to make variations on other people's work.

With bottom-up the constraints, limitations, and natural texture or grain of a medium push into

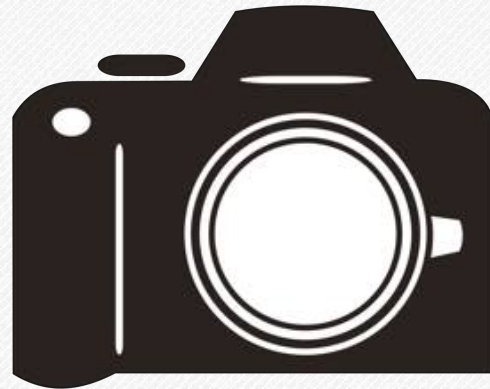
existence results that are not entirely of human origin. The medium becomes a co-creator, an inanimate but unyielding creative partner, arguing for or against ideas thrown its way by what it will and won't agree to do.

A top-down development team strives to create as perfect a realization of their concept as their available platform technology and skill strengths can support. The hidden member of their team is the work that has been done before them, by bottom-up pioneers, and they benefit from the market's feedback on past products.

A bottom-up development team strives to discover as perfect a concept for the platform technology and skill strengths that they have or can learn. The hidden member of their team is the technology itself, and though they don't have the same benefit of borrowing from the past, everyone benefits when they help create the future.

PHOTOGRAPHER'S ALGORITHM

This is the center of design process! Only so much can be figured out ahead of time. Iteration only makes things into nicer versions of what they started as. Make many start options, then keep and refine the best.



Professional photographers, even with the best training, the best equipment, and ample experience, still take many, many more photographs than they wind up keeping. There are too many factors, variables, considerations, and subtleties to simply go someplace, take the “right” photo, then call it a day. They instead take a ton of potentially right photos, then compare them against one another to pick the best to keep.

Likewise, no matter how experienced you are, no matter how good your tools are, and no matter how clear of an idea you have for what you’re doing, in matters of design there are usually countless factors involved, and counting on getting something done the best possible way on the first attempt is unrealistic. It’s not good design to rely on luck as the mechanism for identifying a solid initial effort to iterate further upon. Try minor and major variations, but also try some designs that are

completely different, built from wildly different strategies or initial assumptions from the ground up – whether we’re talking about levels, weapon tuning, game economy, AI scripting, or otherwise.

People with traditional design backgrounds probably take for granted that this process happens in all design. Designers doing product design, package design, t-shirt design, and so on are used to quickly trying out dozens of different ideas and tossing most of them out.

However most gameplay designers don’t have a traditional design background. Because much of our work, even at a prototyping level, often involves a sizable time investment, people get lazy about it and start to act like there’s only time to do one, instead of making several to keep one. If less time is wasted

being overly careful about which direction is tried, and that time is invested instead in just creating multiple different options that are far enough along to be meaningfully assessed, there’s a lot less guesswork involved. It takes some of the luck out of it.

Make several variations. Keep then clean up the best. Throw away the rest.

That was my process for Boom Blox levels, Shotgun Debugger levels, Vision by Proxy: Second Edition levels, Vectorverse levels, and more.





As Fried and Hansson say in *Rework*, “Be a curator... What makes a museum great is the stuff that isn’t on the walls; what makes a museum great is there’s someone saying ‘No’ to things.”

Curate like a photographer.

Don’t make just enough and then find yourself stuck with the worst of it. A good tool isn’t only suited to rapid iteration, but also to rapid exploration, to figure out through experimentation what’s worth further iteration.

No matter how good you are at what you’re doing, the best 40% of your work is better than the other 60%. Working with that plan in mind to edit aggressively can free you up considerably to take chances that you might not take if you always create with a sense that you’ll be stuck with whatever idea or direction is first attempted.

Several of my iOS apps were designed as highly polished versions of carefully selected projects out of my 219 daily experimental gameplay projects. Think about that for a minute: I spent seven months having virtually no social life in order to make that happen, then I filtered out 98.6% of it. But of those three small apps, one led to my series winding up on JayIsGames, one was a finalist showcased at IndieCade 2010, and the other paid my rent for a year and a half.

There are actually similar stories for other developers that led to the

more commercially well-known games Crayon Physics (from a series of 1-month game prototypes) and World of Goo (from a series of 1-week game prototypes).

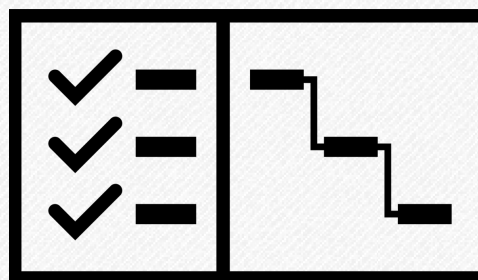
If you're having trouble deciding what to throw away: come up with some simple weighted criteria (these can be subjective values – “understandability,” “aesthetic appeal,” “fairness,” “spectacle” – weighted to give some criteria more emphasis than the others), score each attempt you've made from 0.0-1.0 in each column, find the weighted sum to rank those scores, then throw away the bottom X% of your attempts, where the greater you can make X and still get away with it, the better. It's a handy way to make a rational decision when your gut is telling you that you want to keep all of them for different reasons.

But use the Photographer's Algorithm, and you'll consistently wind up with something much better than you could have

accomplished nearly any other way. It allows direct comparison, in-engine, in-play, between potential results in a way that's otherwise simply impossible to do.

A LITTLE PLANNING CAN GO A LONG WAY

How to best follow a section about the limitations of creative planning? By acknowledging that there are times when planning really helps! It can't ever be perfect, but it's important to not ignore it altogether.



Over-preparation is often a form of procrastination, about how people over-complicating everything, and of course, I frequently advise against starting by writing up 20 page design documents, in favor of making a mock-up screenshot then programming just enough to bring it to life, with at most a one page treatment of gameplay and basic theme notes.

But I don't want to overstate the case! Especially after you've gotten over learning some of the essential skills and ideas, after

you're past the point where you need to follow tutorials to make anything, after you've maybe cloned a game or three and are shifting toward doing your own thing. Suddenly situations will arise where the blocking issue between you and moving forward is the need to sit down and plan a little.

Mistakes will be made.

Over planning is going to happen.

But making those mistakes is the only way to learn how to avoid

them, carefully and narrowly, in your own future. Even if that mistake happens to be overplanning, sure you'll be relearning the same lesson so many others have already learned before, but it's time for you to be the one learning it! And, sure enough, people will still need to learn it after you've figured it out.

When someone gets stuck, whether unable to make progress or unable to get started, it's often a case that there's a step being skipped. Look for that step, something to use as a stepping stone between where you're at and where you're trying to get. Think about the additional process steps that you introduce to any other kind of creative or problem solving process, since videogame making involves both. Brainstorm, concept map, rough draft, sketch, storyboard, diagram of the problem, draw up a map, write enough down until your mind is clear of whatever's holding it up (something an old teacher of mine

called Brain Puking). The sky's the limit on what the step might look like or might be. But consider that a step may be missing, look for what it might be, try anything that comes to mind to see if it unclogs progress and if it doesn't then try something different until you're able to move forward.

The danger of excess planning is usually a result of someone trying to come up with a plan who doesn't have enough context and practice yet for the plan to accurately reflect what realistically can or will happen. The more experience you accumulate though, the better positioned you become to account for your future development a useful, productive way. This includes foreseeing some likely sources of complication in advance and already possessing some strategies and fallback plans for how to keep rolling – even developing a comfort for how it's not always a problem when plans and reality diverge, as long as the

right things are still falling into place.

Excess plans can get in the way of beginners, becoming an excuse to work on something other than the game. Lack of plans can get in the way of more experienced people with more ambitious targets. At some point that's a necessary but difficult transition to go from planning being a misuse of time to planning being a necessary use of time.

Here is where planning really begins to be learned, and not because it's some abstract, artificial, bureaucratic exercise that we're told to do before we're allowed to start the next part. We learn – and this is actually kind of exciting – because you actually can derive real utility from doing it. You need something to happen, but until you jot a few of the right things down, it's simply not going to happen. That's kind of crazy, right?

It may be as simple as writing down a half page plan, first as an unordered list of what's left to be done, then rewritten a second time ordered by which sequence will make the most sense for you going forward.

Just because a plan needs to end up short doesn't mean it needs to be short at first and only ever be short, sometimes like the process of overproduction (or the Photographer's Algorithm) it can help to begin by getting it all out of your system and putting too much down, then going through it to find highlights of what's really worth saving from all that.

And though I keep saying write it down, let me stress that I mean exactly what I'm saying here: you've actually got to write it down. Information in our head is too easily changed every time we think of it. Digital files are far too easily changed, or far too easily ignored by being tucked away where we won't see them since they take up zero physical space.

When I say brainstorming, concept mapping, scheduling, etc., that's really something to be put down on paper. If you're more comfortable working in a digital format – and I know this can sound like heresy in 2014 – print it out and put it where you'll see it. Again because it takes up physical space it'll get in your way, it'll remind you, you can't as easily deny it or easily change it. Trust me on this, if it's the difference between your project stalling out from paralysis, or getting done, then it's a darn good use for one sheet of paper.

Sometimes it's true that in order to make progress, you need to just sit down and put the time and work in, writing code, churning out art and audio assets, iterating on level layouts. Even if there's a problem you're going to run into you need to work on that problem up until the point at which you're stuck in order to ask for help from someone productively! However other times, it can be just as

important to put on the producer hat and work out a list on whiteboard or paper about what's left to be done, and out of that, what should happen, by when, just to know at least what to do next.

There's a reason large teams tend to have producers, and if your small team or solo effort doesn't have a producer, one or more of you is going to need to fill in that role from time to time, ideally, at least a little bit of every day that something is supposed to be making progress on your project.

Feeling a bit overwhelmed at all the options? Not sure what type of chart or graph or process to apply? Then let's keep it simple, here are two specific planning instruments that, used in conjunction, can be used to bulldoze through nearly anything.

(1) Make a high level – meaning non-detailed – list of what's left that needs to be done for the project to be considered finished.

Your goal in making that list is to make it as short and uncomplicated as possible. Don't list any nice-to-haves on it, put those on a second list to, which realistically you'll never look at again ever, because if you find yourself suddenly having oodles of time later to reinvest in the project you'll probably have better ideas then anyhow of how to spend it.

(2) Divide that real list of what's left to be done up into realistic weekly chunks, so that you're not trying to get it all done at once immediately – too much pressure! – nor letting yourself put it all off until the last week – which is just not going to happen! That is your new schedule now. Maybe you had a schedule before. This is now the schedule.

Be flexible about it, swap order of things if it makes sense to do so by bartering between weeks, and if in the process of making continual forward progress it doesn't go quite as fast as you optimistically hoped don't beat

yourself up over it. One of the hard skills you're learning in this case is how to plan and estimate your work more effectively. That's another type of skill alongside doing it.

It's a steady plan to wrap up your game in some form, even if it's not exactly what you originally had in mind. Sometimes that can even wind up better with a more focused game. Or if nothing else, it'll set you free to collect your thoughts, avoid the guilt and lost learning opportunity of an unfinished project, and move on to the next thing in a finite period of time.

Want another simple planning idea to turn to? If you find that when the day is gone that you can't account for where the time went, devise a way to add more structure to your time, and break your weekly goals further into daily goals, or even further into: 1 tangible thing you'll have done before lunch, 1 tangible thing you'll get done in the afternoon,

and/or at least 1 tangible thing you'll get done for the day before letting yourself crawl into bed.

Stuck, or finding you're not progressing on a chunk? You've got to break that chunk down further, you're probably still trying to bite off too much at one time to chew. You'll choke. The metaphor exists for a reason.

Let's try on another metaphor: you've got to be doing the programming, design, and art, just like a road trip is about driving. You've got to be behind that wheel and driving. But making yourself a map, and thinking ahead a bit about how much you'll be able to handle in each stretch, or where you'll be able to comfortably stop, that's a tool and process to keeping your trip or project on course to a destination. If you just sit in the car and hold down the accelerator you're going to run out of gas, almost certainly wind up in no place interesting, or even wreck the vehicle with you and other people in it. If it sounds

irresponsible, it is! Even more so when other people are onboard, planning has to be part of doing.

Don't let unwillingness to find a pen or use a few sheets of paper keep you from making forward progress. It's still important to not get sidetracked into burning energy coming up with fantasy plans that'll never happen, but a short bit of written planning may be just what's needed. Figure out what's left to do, break those chunks into spread out weekly objectives, and when you swap your producer hat back for your designer/programmer hat, have some trust that producer you knew what he or she was doing when the weekly schedule was put together as assignments.

DOING MORE WITH LESS: SHORT VIDEOGAME DESIGN

When developing a hobby videogame without a budget – whether alone or with a small team – it's important to figure out ways to do a lot with a little. Here are a handful of different ways that you can do just that.



Rapid advances in computing, design community discourse, and demographic variation continues to make game development and distribution easier each year, prompting a tidal wave of new games. Although console manufacturers historically limited the number of annual releases, the internet and smartphone app spaces have no such restrictions.

Videogames used to be light on content due to limitations of technology. Then they ran into limitations of budget. The latest

and increasingly dominant limitation now seems to be consumer time and attention within an ecosystem of constantly improving free/cheap entertainment alternatives. Not only is each game competing with the flood of other quality games, either – it also battles for attention against the internet in general and pirated media.

This has prompted unexpected moves like GameStop purchasing Kongregate, and Electronic Arts buying Playfish. Like it or not –

and I certainly have mixed feelings about this – the future of the game industry may look more like The Fancy Pants Adventures than Mass Effect. Even iPhone app budgets are shrinking rapidly, and they were comparatively bite-sized projects to begin with.

As developers, then, it is increasingly worthwhile to strategize about being resourceful – cleverly finding ways to do more with less, until we discover ways to make what we want while spending next to (or ideally) nothing.

In the material that follows I will mostly keep to examples from my projects – not because they show the best ways to deal with minimal resources, but because they show the best ways that I know firsthand. If I knew the design or development of anyone else's projects nearly so well as I know my own, I could focus more on those, instead. (I will, nevertheless, try to reference at least a few

outside games to highlight variations and counter examples.)

CHALLENGING ENDING FOR SYSTEMS GAMES

Topple (iPhone) was not a long game. It has 10 levels, each of which can be played in 1 minute each. 10 minute game, right?



When the game was released – the publisher ngmoco may have adjusted the difficulty since – it took 40-90 minutes of play to complete, due to a mix of luck, strategy, and skill required to advance. In particular the first 6 levels were designed to be a bit easier, with 7-10 being much more challenging.

In a story game, that difficulty spike would be substantially more frustrating. It would create the impression that there's content being kept away from the user. A person buying a DVD expects to be able to see and hear everything from it, and a lot of larger games strive to be associated with film, resulting in an expectation that the user bought and thus deserves to watch all content included. Since Topple is about systems, rather than story content, replaying a level isn't keeping the player from seeing the ending – it's simply playing the game.

TOY WITHOUT OBJECTIVE

Burnit (iPhone) / FireWriter (web) isn't a "game" – there's no score, points, or goal. Instead, it's just a way to draw with gunpowder, then light it (optionally, in Burnit, over a photograph). There's no level 2, there's no plot, there's no



expectation set that it needs more depth or functionality.

Tumult is a similar example. Relative to most iPad apps, there's virtually no content – no animations, no music, no story. Users aren't complaining about its length, though. People like it.

We expect songs to be 3-5 minutes long, movies to be 90-120 minutes long, and television shows to be 22-45 minutes – whether we get them from the bargain bin, pay full price, or watch/listen for free. We similarly expect videogames to be a certain minimum length regardless of price. Tumult avoids looking like it's trying to be a videogame, avoiding those expectations.

Those two are not videogame-like. That's okay, if our goal is to entertain, rather than to make a game. However a toy can be like a videogame.

MEANING OUTLASTING THE GAME

Transcend (iPhone/iPad) can be completed in 15 minutes. It has roughly the same replay value as a book – it's there for review at any time, but won't behave differently than it did the time before. In spite of this, user reviews recommend it.



Rather than focusing on the time spent playing the game in this case, the emphasis is offloaded on making a memorable impression. Unusual presentation, unusual design approach, and a few dozen highlights pulled from over 15 hours of text read from an influential book in the public domain (*Walden*). It ends with a short “song” juxtaposing half of the game’s excerpts over a catchy tune.

My interest was not how the user would think about Transcend while using it, or how long it would take, but rather how and how long the user may think about Transcend after.

Meaning doesn’t have to be dense to outlast the game. Especially if the game is short. This is how I got away with Candy. This strategy also seems to central to Ian Bogost’s satire game Cow Clicker.

HIGH REPLAY VALUE

feelforit (iPhone/iPad) is an artsy project, but mechanically it’s built around a novel interactivity concept rather than a sequence of content. By presenting a repeated, randomly new situation each round, rather than having levels,



the game cannot be “completed”. Each time it’s as good as new.

It’s a skill to be mastered, rather than an obstacle course to be completed. It doesn’t translate naturally into challenges of different difficulty – they’re all equally hard until the skill is learned, and they’re all equally decently engaging, a bit like juggling 3 balls. Had I added two or three “levels” to it, the game would be completable or “digestible.”

(I think this particular distinction, on a massively different scale, was one of the design faults in Spore that previous games from Will Wright didn’t suffer from. In SimCity, the player is fundamentally always doing the same thing, even if at different scales, making there no end to what’s there; in Spore there were – necessarily – a discrete number of different gameplay modes, such that by the time the player was “trained” in all of them, the game felt completed.)

WITH VARIED CONTENT, UNLOCK EVERYTHING

This is a theory of what I should have done, based on what happened. iZombie Death March (iPhone) has 6 level scenarios – each plays differently, with unique weapons, in a distinct setting, and different features. Because the game had a textual story, I locked



the levels into an order: level 1 must be completed before playing level 2, 2 before 3, up to 6. Each level only lasts a few minutes, during which time a random quantity and ordering of zombies attack, of varying types depending upon the level, and in different speed/quantity depending upon difficulty selected.

All 6 levels, therefore, can be completed in order relatively quickly. In that time, players may not even come to appreciate the differences between the levels, weapons, lighting, perma-gore feature, etc.

If, instead, all 6 levels were made available immediately after download, and instead of being a 6-level linear game it was a random zombie shooter with “6 ways to play!” or “6 Zombie Minigames,” I don’t think I would have received as many criticisms about the game being too short.

STRETCHING OUT RELATIVELY LITTLE CONTENT

Alice in Bomberland (iPhone) is a hybrid literature/action mash-up, and partly designed around theories about educational interactivity atoms. It’s a dodging game with 7 types of weapon behaviors to avoid, in front of 8



different backgrounds – yet the game lasts roughly 3 hours.

While this game turned out much longer than the other “short games” on this list, it does not have proportionally larger asset requirements. A few design tricks were used here to squeeze the most out of what’s there.

- Alice is the only animated character. All others are presented as paintings holding weapons during story sequences, which interact with gameplay via attacks fired from off screen.
- Recolored backgrounds. Each of the 8 background images, in addition to appearing in its full color version, also appears in dark (challenging), blue-tint (even harder), and red (final version, most difficult). These required very little programming time to create, but added significantly to the variety in visuals throughout the game while giving an extra sense of proximity to the end.

- Lots of power-up types. Every 1-2 levels, a new item is introduced to the game. Programming the effects for these items was fairly straightforward, and their graphical requirements were also very simple. This spreads out novelty over all levels in the game.

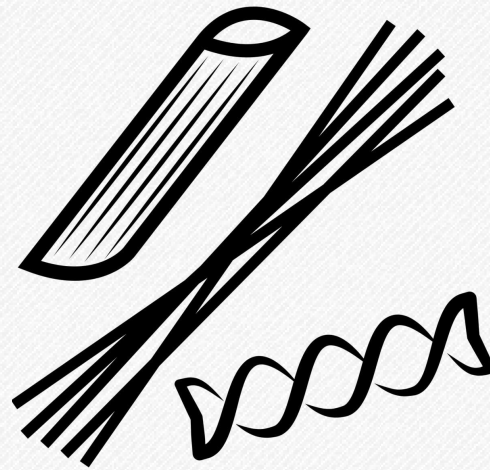
- Unpredictable ordering of level patterns. There are six level difficulties for each of the eight character types, creating a total of 48 levels. Rather than putting character types in repeating order like A, B, C, D, A, B, C, D, A, B, C, D (i.e. always putting a Caterpillar stage immediately after each White Rabbit stage), I instead ordered as A, B, A, C, B, A, D, B, C, D, C (like the distribution of grassy, rails, caves, and water stages in Super Mario World). This allowed me to save certain types until later in the game, include early types throughout until the very end, and avoids a sense of monotonous routine. Different players have different level types

they prefer or hate, leading to surprise “Finally! Another Hatter level!” or “Ooo, a Cheshire Cat level this time” like a mini-lottery after every win.

- While a lot of writing work still had to go into selecting, editing, and ordering excerpts from *Alice’s Adventures in Wonderland*, as with *Walden* for Transcend, the core of the writing already existed and had passed the test of time, saving a considerable amount of time and energy. (In both cases, the copyright on the book text in question expired long ago.)

COLORFUL OCEANS AND CHUNKY SAUCES

Difference in context, style, audience, or appeal means there's no perfect game, but only different kinds that appeal in different ways to different people. Be on the lookout for ways that you're well-positioned to fulfill!



The 2005 book *Blue Ocean Strategy* popularized the distinction between Red Ocean and Blue Ocean. Red Ocean refers to competition with the existing market leaders at what they do best. Blue Ocean refers to making something characteristically different than what market leaders are doing.

The arms race taking place between Medal of Honor, Battlefield, and Call of Duty franchises is an obvious example of Red Ocean strategy. They

compete over a huge audience of known buyers by trying to have more realistic graphics and sound, more single player content, better voice acting, more persuasive ads, etc.

Minecraft or Wii Sports (at the time of its release in 2006), on the other hand, are Blue Ocean. They were not trying to out-Call-of-Duty the real Call of Duty. They were producing something vastly different for a difference audience, a sizable portion of which probably doesn't care about Call

of Duty at all. They didn't win for having the most impressive graphics and sound, they won by each doing really well at things that games like Call of Duty never even attempted to do.

A company isn't going to acquire the combined audiences by simply integrating more aspects of Minecraft and Wii Sport into a warfare game. Many of these preferences are mutually exclusive – those of us that like Minecraft enjoy it as much for what it isn't as for what it is. Likewise, giving Wii Sports a Michael Bay treatment of huge explosions and machine gun fire would very likely turn off a massive chunk of the audience that enjoys that game.

Of course, now that Wii Sports has been out for 5 years, releasing games like it is much more of a Red Ocean than it used to be. When Sony's PS Move and Microsoft's Kinect eventually attempted similar games on other consoles in the current generation, they were grasping for whichever

players haven't already experienced enough of the original on its cheaper platform, or they were hoping to win over those same customers on claims of improving upon the formula, which is a clear sign of Red Ocean.

Chunky Tastes

We're going to hard switch now from Red and Blue Oceans over to spaghetti sauce. It's a similar, though subtly different point, that has to do with finding and appreciating Blue Oceans.

In case you haven't heard Malcolm Gladwell's TED Talk on spaghetti sauce, it's well worth the 17 minutes:

http://www.ted.com/talks/malcolm_gladwell_on_spaghetti_sauce.html

The craziness that happens when fans argue about whether Battlefield 3 or Modern Warfare 3 is "better" is that they are arguing between whether plain or spicy spaghetti sauce is better. Despite surface similarities, the games are different in their pacing and

appeals, and resonate differently with everyone based on whatever styles they happen to prefer.

A few snippets that are especially relevant to hobby and indie videogame development:

Now, did he look for the most popular brand variety of spaghetti sauce? No! Howard doesn't believe that there is such a thing... And sure enough, if you sit down, and you analyze all this data on spaghetti sauce, you realize that all Americans fall into one of three groups. There are people who like their spaghetti sauce plain; there are people who like their spaghetti sauce spicy; and there are people who like it extra chunky.

And of those three facts, the third one was the most significant, because at the time, in the early 1980s, if you went to a supermarket, you would not find extra-chunky spaghetti sauce. And Prego turned to Howard, and they said, "You telling me that one third of Americans crave extra-chunky spaghetti sauce and yet no one is servicing their needs?" And he said yes!

And Prego then went back, and completely reformulated their spaghetti sauce, and came out with a line of extra chunky that immediately and completely took over the spaghetti sauce business in this country.

And over the next 10 years, they made 600 million dollars off their line of extra-chunky sauces.

Gladwell also touches on the importance of recognizing that people don't know what they want. You can't just ask them. People can't even properly say what they want when they have had it many times before, and given that, they are even less likely to be able to articulate what they want but have never had.

Assumption number one in the food industry used to be that the way to find out what people want to eat — what will make people happy — is to ask them. And for years and years and years and years, Ragu and Prego would have focus groups, and they would sit all you people down, and they would say, "What do you want in a spaghetti sauce? Tell us what you want in a spaghetti sauce." And for all those years — 20, 30 years — through all those focus group sessions, no one ever said they wanted extra-chunky. Even though at least a third of them, deep in their hearts, actually did.

People don't know what they want! Right? As Howard loves to say, "The mind knows not what the tongue wants."

It's a mystery! And a critically important step in understanding our own desires and tastes is to realize that we cannot always explain what we want deep down. If I asked all of you, for example, in this room, what you want in a coffee, you know what you'd say? Every one of you would say, "I want a dark, rich, hearty roast." It's what people always say when you ask them what they want in a coffee. What do you like? Dark, rich, hearty roast! What percentage of you actually like a dark, rich, hearty roast? According to Howard, somewhere between 25 and 27 percent of you. Most of you like milky, weak coffee. But you will never, ever say to someone who asks you what you want that "I want a milky, weak coffee."

That speaks to the importance of prototyping, and early testing (with direct player observation of their responses, rather than simply trusting what they say afterward). People won't know if they'll want it before it's made, even if the game could be perfectly and completely described in advance. Our behavior is far more telling than our verbal guesswork when it comes to knowing what we want, and that means getting the idea

working and in front of people. Howard's tests weren't surveys about how chunky people wanted their sauces – they were rigorous taste tests with more than 40 variations of fully prepared ready-to-eat sauce. (I used a similar process when responses to my 219 daily prototypes led to my iOS apps Tumult, feelforit, Burnit, and iZombie Death March.)

Returning to the main point, but stated another way:

Mustard does not exist on a hierarchy. Mustard exists, just like tomato sauce, on a horizontal plane. There is no good mustard or bad mustard. There is no perfect mustard or imperfect mustard. There are only different kinds of mustards that suit different kinds of people.

At the end of the day, at least in terms of design decisions (as opposed to technical quality, such as not crashing), games don't exist on a quality hierarchy.

It doesn't matter whether the people that like some other type of massively popular game also like your game. Audiences looking for

the types of experiences that huge companies can deliver to a high level of polish already are quite satisfied with what they are getting (except to the extent that they occasionally feel angry over other people having different tastes). The preferences and interests of smaller groups – groups perhaps too small to be of interest to major companies – are equally legitimate and important.

Maybe the outcome of a particular project will turn out to be chunky, a huge category that previous developers didn't even realize existed, as has been the case for Minecraft or Wii Sports (the latter example which I love, in part, because it's solid evidence that Blue Ocean isn't just for indie developers).

Or maybe it will simply be a smaller audience, like finding one of the other 36 spaghetti sauce styles on the shelves. Such an audience may even appreciate it more because they have such a

hard time finding anything that suits their niche tastes.

Vision by Proxy: Second Edition has been played 6.9 million times, whereas Transcend has been played fewer than 10,000 times. However I have had people go out of their way to contact me about how much they enjoyed Transcend, eager to talk more about the game, whereas the same has simply not been true about Vision by Proxy: Second Edition. I suspect that difference is partly because VbP:SE is more recognizable as a videogame, and fills a more commonly understood niche, as compared to the much smaller number of people that found Transcend suited to their preferences – who understandably might have trouble finding many videogames that scratch that type of itch. I don't regard either project as better than the other, a luxury that comes with neither being a commercial undertaking (I invested no money in the development of either, and the

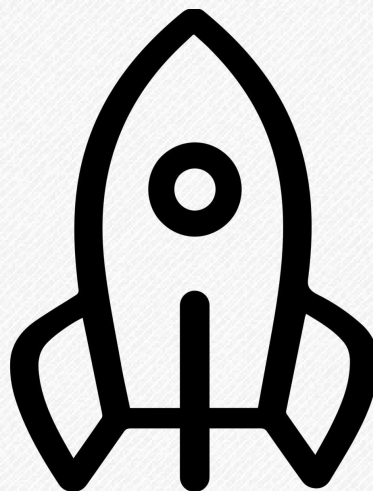
excellent work done by teammates on Vision by Proxy: Second Edition was likewise part of a group volunteer effort). They are simply “different kinds... that suit different kinds of people.”

As a hobbyist, I think that’s a healthy attitude to maintain, and I think it’s also a useful lens to put the work of other developers, both large and small, in context.

[Credit where it's due: I would like to thank thatgamecompany's Robin Hunicke for introducing me to the idea of applying Red and Blue Ocean thinking to independent games. I first heard about it briefly in an exchange during our one day of overlap on the Boom Blox team back in 2007, but it's a way of thinking that has stayed with me and affected much of my work since.]

GENRES AND CONVENTIONS: KNOWN PATTERNS OF WHAT WORKS

While it's good to pick fights we can win, in deciding what to make it's also essential to never lose sight of the real value in adhering to some recognizable genres and conventions. People can't crave what they can't "get!"



HATING ON GENRES

Game designers sometimes speak poorly of genres. I strongly suspect that this same dialog has occurred and continues to occur in all other creative communities, from novelists to screenwriters to visual artists, though of course game designers are the community that I'm focused on. Closely related are the complaints about all games seeming too similar, that there isn't enough experimentation.

In reality there has been a ton of experimentation in terms of creating non-formulaic videogames – for decades – since the dawn of home computers, and even as published (though now largely forgotten) games on early videogame consoles. Such games likely outnumber those titles that fit easily into genres. Whether or not it has already been the case historically, it has undoubtedly come to be in the tidal wave of noncommercial or small scale (ad supported etc.) game

development within the rise of web games and smartphone apps. Videogames have basically been in a constant Cambrian explosion since their inception.

FINDING LESS FORMULAIC GAMES

The notion that there are and have ever been only a handful of types of games is a side effect of selective awareness. In particular, it arises from only giving attention to those games that received chart-topping commercial success or extraordinary critical acclaim. Both of those tend to be disproportionately genre works. Want to find the more experimental videogames? Look past the Top 10 lists, into the greater catalog of games with a metacritic score in the 60s and below, or even more so among games too small to be reviewed or listed on metacritic, and you'll find a lot more of them.

That domain has plenty of failed shots at the top 10 list, too, but within that low to no score space the weird projects are much more

common. When there's news about a genre-breaking game among the critic, developer, and player communities, it's not simply because it's genre-breaking – it's rather because it succeeded on top of being genre-breaking, unlike the rest of the genre-breaking shovelware in the bargain bin and buried countless pages deep on web game portals.

Why is it that so many people think and say that they don't want genre works, but then go buy them and tend to rate them higher? Let's take a brief look at how and why genres work in videogames.

WHY GENRES GET ATTENTION

Some of the angst against genres, at least among indie developers if not players, perhaps inadvertently overlaps the Red Ocean / Blue Ocean distinction, which was covered in the previous section. Lazily painting with broad strokes, we might consider genres as Red Ocean basically by virtue of having been established enough to gain a

distinction as a genre, while work that is without a clear genre might seem Blue Ocean by the opposite: it's less contested space because it hasn't yet been identified as lucrative to iterate on/within.

It makes sense, I think, that once a particular style of product has clicked with the public and a clear demand has been demonstrated for it, then and only then does Big Publisher move in to try to outdo everyone else at making the biggest, most impressive version in that category to capitalize on that trend. This of course folds back into the visibility of the games that fit squarely into genres relative to games that don't: games in those clearly understood genres connected to substantial consumer demand are what can get the most development funding, marketing budget, and iteration in the form of companies battling out every variation and sequel they can think of in a struggle to get that formula just a

little more right than the other and previous efforts.

There's more to the success and frustration over genres than that, though. I think it's essential too to not doom ourselves, as solo and small team developers, to obscurity by ceding genres to big studio games, when the two really aren't quite a 1-to-1 mapping to Red Ocean / Blue Ocean.

"GAMENAME"-CLONES

Where do genres come from? Generally, someone doesn't deliberately invent a genre and coin it. Companies and people simply make a game, and if the game does very well then other people clone and vary it... until those clones and variations constitute a genre.

At some point after enough of a game design's derivatives begin to fill up the retail shelves or front pages of review sites, it starts to seem strange to refer to them anymore merely as "GAMENAME"-clones. First-person shooters were Doom

clones, platformers were Mario clones, first-person puzzle click adventures were Myst clones, overhead turn-based RPGs were Final Fantasy clones, overhead real-time combat adventures were Zelda clones, head-to-head fighting games were Street Fighter clones, while match-3 puzzles were Bejeweled clones and open world driving city games were Grand Theft Auto clones. Those last two still are referred to as such in some circles, these being a comparatively more recent. Note too that in none of these cases are those actually the first game of that type, they were just the first breakout success, or otherwise the first highly visible, such that those games defined what the projects following their formula afterward desired to emulate or improve upon.

DOWNSIDES OF IGNORING GENRES

I did much of my personal exploration into questioning genres with the InteractionArtist series of experimental gameplay

projects spanning from 2007 to 2008, when I developed mostly genreless digital interaction prototypes daily for seven months. Aside from a few iOS apps that my favorite few led to (Tumult, feelforit, and burnit), one of my main takeaways from working on the series was to better appreciate the value of genres and conventions to developers and players alike. A bunch of those prototypes, anything beyond the most simplistic, left most visitors simply scratching their heads, literally unable to even make sense of what was in front of them. Oops.

There's only so much learning, attention, and new consideration that a typical player is willing to give to a new amusement activity. Building upon a player's prior knowledge and experience can enable more depth or richness with considerably less ramp up. Expectations embedded into genre distinctions are one way that a player's cognitive load can

be, if not necessarily lightened, then at least focused on what makes a game special or different, atop a central framework of interactions and events that the player already understands how to navigate and operate.

RECOGNIZABLE TRANSFER OF SKILL

I have a particular interest in pinball, so for sake of illustration, think of pinball as a genre of electromechanical games. Once someone knows how to play even one machine pretty well, even if there are many scoring details and artistic differences between pinball machines, their basic flipper techniques can largely transfer to other tables. Unambiguous visual clues in the machine's physical structure and playfield layout communicate to the player whether an arcade device that they've never even seen before is also a pinball machine. If it is, then they know even before playing that it's like something they've liked before, something they're capable of doing well, and

something they have some head start on understanding.

The same can be said for head-to-head fighting games, RPGs, platformers, first-person shooters, real-time strategy games, and virtually any other videogame genre. People can look for visual and other clues to know in advance whether it's something similar to what they've enjoyed before, and if so, when they start into it they already have an understanding of how it works and what to do with it. They can get right into the game. That's a tremendous advantage over the many forgettable videogames that strive to eschew (or what one might optimistically reframe as "attempt to start a new") genre.

IN TRADITIONAL NARRATIVE

As another way of thinking about the above, genres in videogames serve some of the same function that genres do within general storytelling: a set of established conventions to serve as a background, any deviation from

which can be a source of meaning. If the sheriff in an old western is the bad guy, that's interesting. If the aliens in a sci-fi movie are less advanced than humans, that's interesting. In the same way great games are often firmly fit within a genre, so that for the most part we can count on understanding what's going on, except with one or two central twists (the player controls gravity, the player controls time, the player can possess enemies...) and polish and iteration evident in the execution. That frees the mind to focus on what makes it interesting or different, without needing to exert much brainpower for understanding the core of what's going on.

COMPARISONS IN MUSIC AND ARTWORK

However it's not merely a matter of what people have practice in, or what knowledge they've already learned. That can make it seem much more arbitrary than it is.

Genres and conventions reflect patterns that have been found to

be well received by many people, and perhaps more importantly, particular people (the same many who keep coming back for more). Consider music or images. Most of the technically possible arrangements of notes do not constitute anything even remotely musical, and likewise most of the possible arrangements for color on a 2D plane produce something not just nonsensical but uninteresting (as opposed to nonsensical and interesting, which of course actually describes a great deal of rather famous art). Tell me however whether music is techno, hip hop, rock, or country, and I immediately have some notion of whether I'm likely to enjoy it, since despite there being such great variety within genres it's of a significantly more limited scope than genre-less sequences of sound. Likewise tell me that an art exhibit is surrealist, cubist, impressionist, splatter painting, a graphic novel illustration, etc. and even without knowing more

specifics about the art, I can form some sensible guess of whether it's something I'd at least be interested in seeing.

Most of the possible interactive graphical programs that could be written do not constitute anything that even remotely works as a videogame. Whether a videogame is 25 kb, 25 MB, or 25 GB, most other possible configurations of those bits go beyond meaningless into the realm of unusable. But tell me it's a platformer, first person shooter, third person adventure game, or puzzle game, and I know enough from past experiences with games we assign those labels to to make a decision about whether I might like it.

BAKERS IGNORING CHOCOLATE AND CAKE

The analogy I find most useful for thinking about genres for videogames though isn't how genre applies to music, or image, or novels, but instead the parallel to how genre applies to food. At the core of gameplay, rather aside from narrative bits and the

environmental art or audio content of the game, is a relationship between input, movement, and camera that creates what Steve Swink labels "Game Feel."

Expectations for how this does or doesn't work out varies between genres. As with those previous examples most of the theoretically possible combinations flat out don't make sense, leaving islands of flavors that one player community or another might tend to prefer.

Imagine a chef trying to combine flavors and food preparation steps in a truly random way in hopes of producing food anyone would actually want. Consider that process in contrast to, for example, making a cake. Or a muffin. Or a cookie. Or a hot dog. Or a donut. Or french fries. Or a burrito. Or sushi. Or falafel. These are all recognizable food concepts, an incredibly narrow slice of the possibly mixtures of ingredients and preparation methods. People can order it

knowing what to expect, and have some sense for how to judge whether it's a good or bad version of what it's aiming to be (a good french fry, for example, would be a very bad cookie). Their value to people however is not merely that they're recognizable or that because they're recognizable they can be judged meaningfully. But because out of the however many millions of possible foods created throughout history, these templates survived while the countless other attempts promptly went extinct, not because anything in particular went wrong, but because so much has to line up just right for a recipe to really work. So when a pattern that does work well gets discovered, it's only sensible for many creators to deliver on variations of and masterful attempts at making those. It is similar for consumers who know what they want from past experiences and go looking for just that.

To some degree – and it's not quite the same thing, but I think close enough to warrant consideration – complaining over a game designer working within a genre would be akin to complaining over a baker working on bread or a pie instead of actively trying to rethink and reinvent or challenge what baking is. People want things that bakers make, there's still plenty of room within those “genres” for creative interpretation and for the work to come out amazing or inferior, personalized or generic. Most random other combinations of preparation methods and materials at the baker's disposal would be very likely to yield things that even given time would be incredibly unlikely to become an acquired taste, especially alongside recipes and patterns that have evolved and been refined for so many generations.

6



For videogames that have levels, the art of designing a good level is as much a skilled craft as music composition, art, writing, or programming. Some game levels aren't even spaces, but are instead just tuned numbers: waves of enemies, or odds of different puzzle pieces falling!

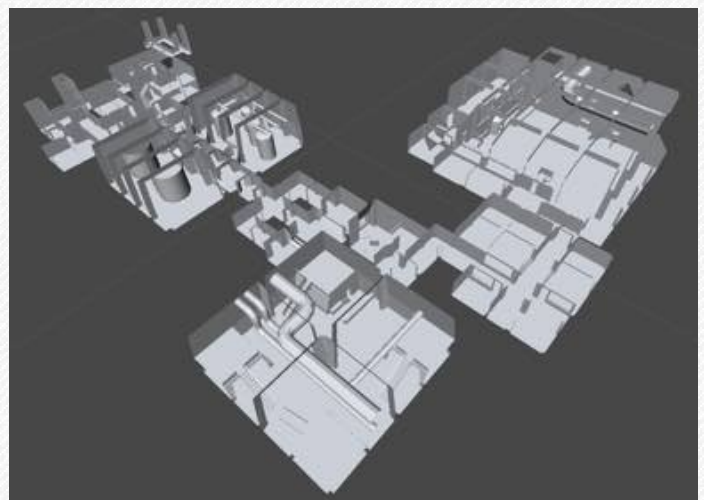
LEVEL CREATION

ANTI-DESIGN / BACKWARDS GAME DESIGN IN GOLDENEYE

Because design involves so many different options, it's sometimes helpful to take a case study approach: find a specific real example, read up on and/or study its parts in very close detail, and try to extract lessons from it.



I've been studying GoldenEye recently—not the newer remake, but the old 1997 game from Nintendo 64. Rare put it together, and Martin Hollis was the producer. Back in 2004, long after a lot of people forgot about the game, Martin Hollis did an interview about the game and its creation. In it he reveals a bunch of interesting facts about how it was originally inspired by Virtua Cop in certain ways – as he puts it, when you hold down the aim button it kind of goes into Virtua



Cop mode – or various other tidbits, but, if you're interested in that full interview check out:

<http://www.zoonami.com/briefing/2004-09-02.php>

But there's a few very specific gold nuggets in there that I want to pull to the surface and share

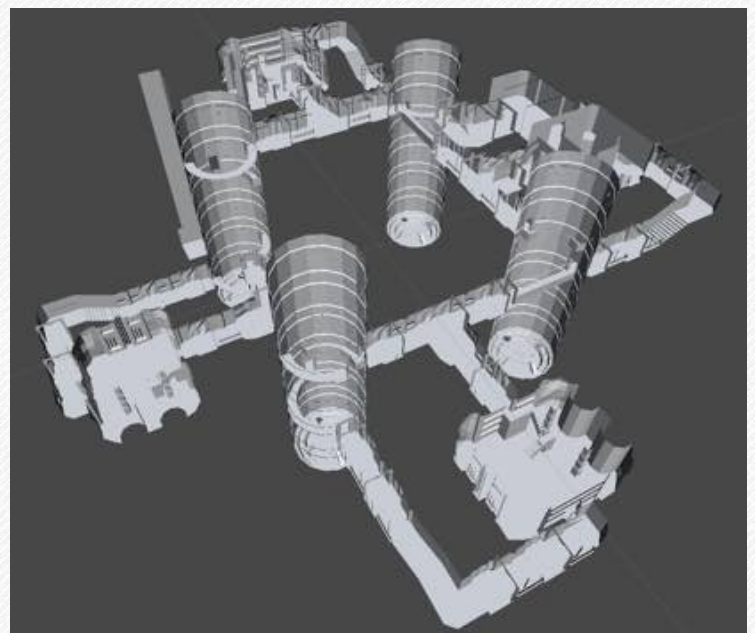
today. Those have to do with the ideas of backwards game design, or anti-game design as he calls it in another place.

The excerpt I'm referring to begins as follows:

"One important factor was this. The level creators, or architects were working without much level design, by which I mean often they had no player start points or exits in mind. Certainly they didn't think about enemy positions or object positions. Their job was simply to produce an interesting space. After the levels were made, Dave or sometimes Duncan would be faced with filling them with objectives, enemies, and stuff. The benefit of this sloppy unplanned approach was that many of the levels in the game have a realistic and non-linear feel. There are rooms with no direct relevance to the level. There are multiple routes across the level. This is an anti-game design approach, frankly. It is inefficient because much of the level is unnecessary to the gameplay. But it contributes to a greater sense of freedom, and also realism. And in turn this sense of freedom and realism contributed enormously to the success of the game."

What interests me about this, to

provide contrast to how game design typically occurs for level design in, say, a shooter or a lot of types of games, is you might first come up with a beat doc of what you expect the player to do in what order. He's going to collect these documents, he's going to get this key, he should blow up a helicopter, and then he's going to escape the base. And then you would design the level around those objectives. You would provide a space in which he can complete the first objective, and then you would assume he would move on to the next and would complete some objective in the next segment of space you give



him. You might give him another room with someone to battle, and he progresses past that. In each case in a lot of games, and we still see this a lot of the time, once you complete an area or complete the objective for that area, you never go back.

Instead, what Martin Hollis had his level designers do for Goldeneye back in 1997, or I guess rather development in 1995 or 1996 or so, was to just focus on making an interesting space, to make these spaces different from one another, to make them compelling to navigate, to make them all have a unique look and feel. They weren't supposed to be worrying about where the player started, where the enemies were supposed to go, where the objectives would take place—what the objectives even were! Their objective was to put some rooms together, put some hallways together. Build a space. And then we'll worry about how to populate it with enemies and objectives and items later.

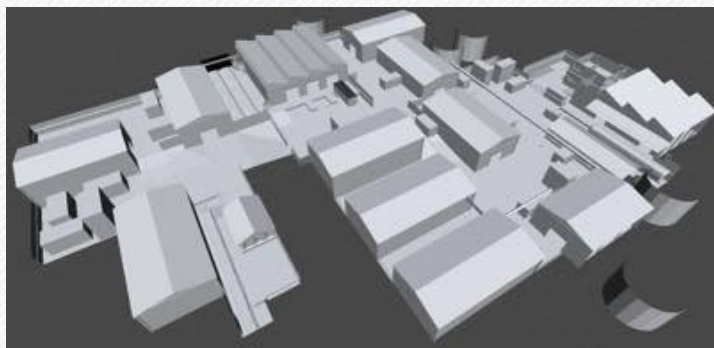
This is, as he calls it, anti-game design. He's beginning by focusing on making an interesting area then populating it, as opposed to the other way around. He uses these loaded words, saying that this approach is, "inefficient," "unnecessary," he calls it "sloppy," "unplanned," though of course he's not saying these things negatively. I want to call attention to the fact that without using this approach, they would not have gotten those same enormously successful results, the sense of realism and freedom that come from having spaces that are so interconnected and tied together, having so many optional rooms off to the side, having so many different hallways that don't wind up being used. That was a product of the anti-game design approach. So if that's inefficient, compared to a usual process that doesn't produce as good of a result, then I would argue that it is in fact the most efficient way that will yield the result that's desired.

Something to keep in mind: is it sloppy? Is it unnecessary? Or is it the way that this sort of good work happens?

This approach actually extends beyond level design. As Martin Hollis mentions later in the interview,

“We milked the Bond universe in many ways. For example gadgets, I compiled a list of about 40 gadgets from various Bond films, most of which were modeled, and then Dave and Duncan tried to find levels where we could use them. This is backwards game design, but it worked very well. These models were the game design; there was very little written down on paper. And the models were researched and milked extensively. And, importantly, they all gelled together very well. These things helped to make the gameplay and the game style what it is.”

So it wasn't just that they built levels without plans for what would happen in them. They modeled and textured 40 different gadgets before they really had specific plans for how to use these things, for which would be used with the others, for which would



appear in which levels. They focused on what are some cool Bond gadgets, and then how can they shoehorn these in after they've already been made, and some of them didn't make it. In the data for the game, there are a number of objects in the memory which never wind up used, not even through cheat codes, and have been pulled out partially but not entirely, because as development progressed they realized they didn't necessarily need all of these. But they have them there to pull from.

He mentions that they all gelled together very well. It's no coincidence of course that when you pick out 40 James Bond gadgets, they all gel together very well, because he's pulling from a very specific source of intellectual

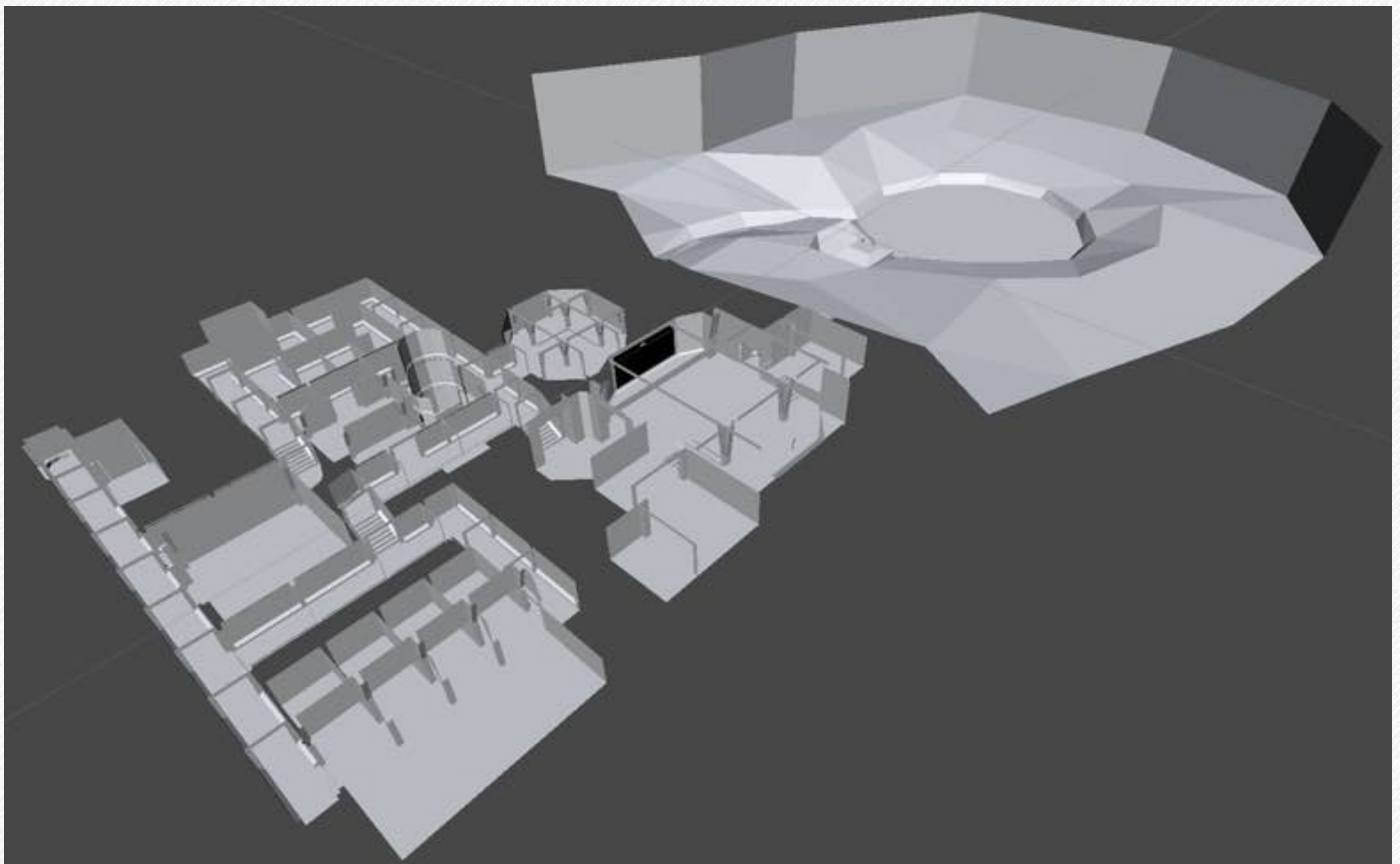
property, that has evolved for a very long time, since the creator Ian Fleming came up with the character for James Bond. Since then every studio, every producer that has touched the property has contributed ideas to it based on what looks cool, what will be neat, what's a great setting, what are some cool weapons or gadgets to introduce... and so that gave them a lot to pull from, in the same way that the Star Wars universe has a lot to pull from, the Indiana Jones universe has a lot to pull from, the

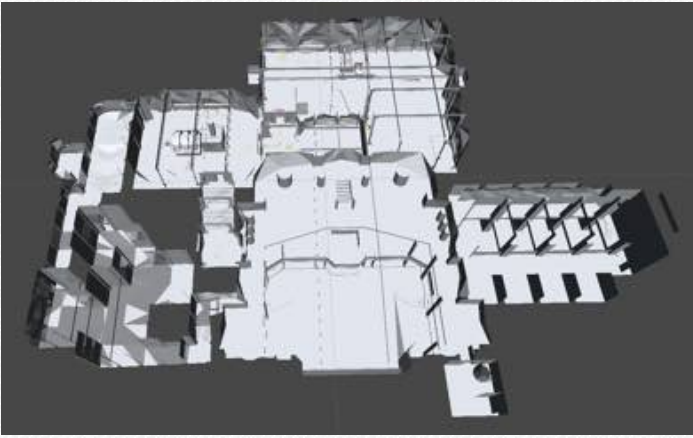
Lord of the Rings has a whole bunch of dynamics about how that world works and how those pieces fit together. These gadgets all gelled because of course they're mostly James Bond gadgets.

The gadgets were inspired by the James Bond films, but so were the spaces, obviously.

As Hollis puts it,

"Karl constructed levels based on the film sets, which we visited several times. And Bea constructed characters based on the photos of people and costumes we had."





So think about this – they’re really utilizing Hollywood film set design, Hollywood film costume design, in their game, by adopting those patterns. Although those aren’t just built off of skilled Hollywood craftsmen, they certainly played a huge role in it. Those were scoped out real locations in Cuba, and in Russia, and so on. Likewise the costumes were based on, in many cases, real army outfits, and the weapons are all based on real guns. This keeps it grounded in a complex cultural, very realistic environment in which the guns and the costumes and the settings are based on something recognizable in the real world.

Anyhow, there are many ways that this can be applied. It doesn’t necessarily mean that you’re

making your levels before you know what to do in them. It doesn’t necessarily mean that you’re making objects before you want to do something with them. It also doesn’t necessarily mean that you go back and write a full complex lore like James Bond’s universe before you start working on your game to ensure that everything will gel. There’s smaller ways that this can happen, too.

David Jones, the guy who created the Grand Theft Auto series and later worked on Crackdown, also created the Lemmings game. This was a classic game where you had these little suicidal characters, and you’re trying to real-time strategize them to block each other and build bridges and build stairs, to make their way to an exit. And so the story goes, that derived from sort of a bet David Jones had with another developer about how small of an animated character he could make, that that small character became the Lemming. It’s a case

where he didn't start with the game's idea, he didn't start with the game's story, or arc, or events. He simply started with a character sprite, and then found a way to build a game to use that.

We followed, of course to much less fame and success, a similar sort of approach with Zylatov Sisters. Before anything else about the game existed, I was modeling the little Zylatov Sisters, or drawing them, just to practice small pixelated animation. Once I had one of those drawn, and color shifted her and got different hair color from her, I decided we could make a co-op little game with guns. That turned into a game for us.

Think about backwards-game design, anti-game design, or call it whatever you prefer. It's simply the idea of

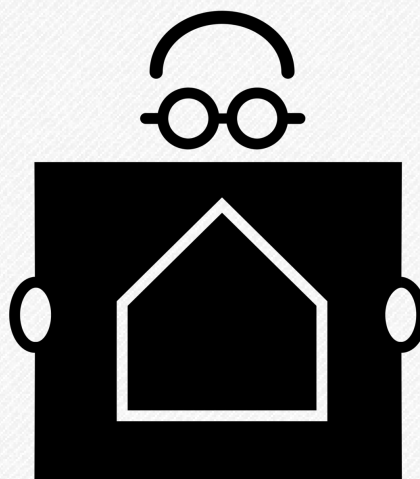
starting by building something that's neat or appealing in its own regard - be it a model, as a space, a character, a sound effect, whatever form yours might take - and then work around making that material fit. For once, try worrying only after you produce the content how to fit it into the game. Different processes tend to yield different results, so this is likely to lead you in a different direction than your usual methods.



Zylatov Sisters by VGDev, available to play free in-browser. For more information about this game see Chapter 10 Section 2, "Reflections on Three Specific Hobby Game Projects"

LEVEL DESIGN CONCEPTS

Certain patterns have emerged in my experiences designing spatial levels for a wide variety of games - whether navigational puzzle platformers, action overhead, or otherwise. Can you try these in your game?



Philosophies

Every decision in a level's design is a conscious act by the level designer.

Levels aren't made by placing walls; levels are made by planning. Once a level developer is accustomed to the toolset at hand and the underlying game engine, emphasis shifts from placing "walls" to placing "rooms," and from placing "enemies" to designing "encounters."

Odds are, you've never played more than one published title that used the same underlying level design philosophy. What works in one game, given its AI, weapons set and player interface generally does not translate well to another title in the same genre. Goldeneye N64 levels make poor Doom levels; Doom levels make poor Unreal Tournament levels; Mario levels wouldn't work for Sonic and Sonic levels wouldn't work for Mario.

Why bother with level design philosophy at all? Why not just make maps that are different every time? Just like there are conflicting philosophies that don't work well between games, for every game there are some overriding points that can be kept in mind to make the most of the engine. Here we'll look at a few particularly different ways of thinking about level space, since by separating them out, we're better able to switch between them deliberately as best fits the current videogame project's needs.

ARCHITECT'S DESIGN

Some games focus on environmental realism, to the point that most of the levels are designed to feel like they are almost incidental to the gameplay experience taking place there (ex. Hitman, Rainbow Six). For these games, most of rooms, hallways, and open areas feel like they were laid out without special emphasis for the player start, ammo and health boxes, or enemy placement

locations. An architect by training is most likely to design this type of map, so we'll call it the "Architect's Design." It provides a strong sense of immersion when it's done well, since real buildings aren't laid out linearly for mission objectives, but it can make for awkward flow that confuses first time action players.

FIREMAN'S DESIGN

Other titles focus on flow of action. Halo is the unmatched example of this design strategy, using Fireman's Design to compensate for the disarray of the battlefield. The player is rarely left wondering where to go next, since there are typically shots, yelling, and action taking place where he/she should go. We'll call this the "Fireman's Design," since it results in the player rushing from point to point to "put out fires." This requires a considerable amount of event scripting, and doesn't leave much of an opportunity for the player to rest. This risks hurting replay value by making interesting

things happen predictably the second time around, but it can offer an extremely cinematic experience (ex. Medal of Honor, Call of Duty franchises).

CURIOSITY LURE

Some games lure the player around via exploration. Tomb Raider and Descent both relied at least in part on this “Curiosity Lure” (the player’s left thinking “maybe this pathway leads to the exit?”). Without careful attention to attractive landmarks in the distance, and clear visual distinction between different rooms, it can lend itself to arbitrary map layouts, leaving the player wandering in cycles through corners for the next area to search through. Tomb Raider, for the most part, succeeded in doing this well, whereas Descent (unfortunately) did it very poorly. In a more action-oriented game, the wandering is made even worse by going frequently through areas well after the interesting part of that genre

(the enemies!) have been eliminated.

REVERSE BREADCRUMB

The name of this technique comes from the old Brothers Grimm fairy tale, Hansel and Gretel, in which a young boy and girl leave a trail of breadcrumbs behind them to find their way back home. It’s “reverse” because, in this method of level design, breadcrumbs are scattered everywhere by the game’s designers, and the player finds their way to unexplored areas by running toward and picking up any not yet claimed.

id Software used Reverse Breadcrumb in Doom, but it has been around at least since Pac-Man, and still appears in modern FPS games. Reverse Breadcrumb is a technique that involves lacing corners, hallways, side rooms, and action areas with cheap, low-value items of cumulative worth. Boost of tiny value, such as +1% health, +1% armor, and little ammo packs are great for this, though in some densely-populated action games

where the player has been is just as easily marked by where bad guys no longer roam.

Besides giving you as the level designer an instrument of instruction to show the player where to go, it provides an immediate visual test for the player to mark off areas he's/she's already visited. If there are no more items (or weak bad guys) in a room with several branching hallway exits, the player only has to explore whichever hallways still have items to find new areas of the map.

This approach constantly rewards the player, and leads to most or all of a map's areas being explored in turn. Care needs to be taken in a map designed with Reverse Breadcrumb to minimize the depth of dead ends, to avoid the player winding up stranded without cheap items as hints.

ARENA TRAPS

Although Painkiller is among the few recent commercial games to exploit this design strategy, games

like Robotron, Mega Man, and Zelda established its place in history. The concept behind "Arena Traps" is to have the player fight battle after battle in isolated architectures. This avoids player's using kill zones to take advantage of deterministic AI. It implies that the world has an overriding, malicious intelligence manipulating the player's environment, but that works so long as the story takes the player to an evil dungeon, a trapped temple, or alien den.

PUZZLE BASED

Jumps, keys, physics engine exploits, and remote switches or time trip wires dominate puzzle based games. It's rare to see an entirely puzzle based game anymore, but some degree of puzzle is more likely than ever to find its way into every game on the shelf. Prince of Persia had a few undisguised puzzles in the story, as did Quake II and Tomb Raider. Portal has more recently repopularized puzzles in the form of intricate navigation.

DISGUISED LINEAR

When the player is strung from one location to the next, but they feel like it's their idea each time, then the level design is Disguised Linear. Done right, this describes a map that plays linear but doesn't feel linear. Max Payne and Half-Life are the best-selling games with this structure. It can significantly detract from replayability, and can also raise some issues of the designer having more fun than the player; if the player doesn't get to decide where he/she will go next, and instead the level designer does, who's really playing? On the plus side, it typically means the player won't get lost, and the emphasis of the gameplay is on action or platform/key puzzles rather than exploration.

A non-linear level structure (like one with a centralized hub and many hallways) can quickly be turned linear by use of keys, remote door switches, and other order-of-play constraining

mechanisms. The workaround to avoid this problem is to take a Mega Man boss weapon approach, so that although the order of action is up to the player, there's an ideal order that the expert player will take to optimize level performance. For these types of unenforced Creatively Linear level designs to work, the player must be given clear clues or signs of what is to be found in different directions – perhaps by imposing architecture, detailed red warning patterns on the floor, or puddles of damaging green goo.

HYBRID

Most commercial games don't follow any one formula. The most interesting level design philosophies are those that manage to take the elements that work from a variety of design angles. Don't be afraid to experiment, but when you do so, you'll probably want to test it in a smaller map before you build it in as a crucial point of a larger map.

Invent some terms, like those used here, and categorize your thoughts into discrete concepts – it makes it easier to re-use the ones that turn out well, easier to avoid re-using the ones that turn out poorly, and easier to mix new combinations of what worked well to generate more intricate levels for later parts of the same game.

Rapid Prototyping Methods

NODE NETWORK

The scientist's approach. Begin by drawing a few circles on a blank page to represent major rooms, areas, hubs or hallway intersections. Draw a few lines between these nodes, until everything is connected in at least 1 way. Next, put your mind to thinking about how you'll run the player through these loops – item lure? Strictly ordered key puzzles?

Advantage: Fast way to sketch out how your level can have complex interconnectedness, without holding yourself back to conventional orthogonal hallways or linear path cutting as you might

be tempted to do in most level creation environments.

Disadvantage: Leaves a LOT of unanswered questions about the level's theme, and the layout of rooms. This prototyping definitely should be used only in conjunction with another method, such as Pipelining or Area Sketching.

DIVISION OF SPACE

The builder's approach. Begin on a piece of paper with a rectangle, pentagon, or a more interesting shape, like a giant hand (Doom, map E3M2 did this), and then recursively divide that space into rooms.

Advantage: Can create a more realistic building space (no wasted area between rooms or halls). Doing it freehand on paper then transforming it into your game's level format reduces the tendency to work strictly orthogonal or on-grid.

Disadvantage: Leads to a map that is potentially too visually cluttered. If the overriding shape is not

broken carefully, it can also yield a map that feels too contrived.

PIPELINING

The mechanic's approach. This can be done equally well in most level building tools as it can on graph paper, if you're sufficiently comfortable with its environment and tools. Basically, form the level out of hallways, placing important items and keys in intersections or corners. Go back and bloat certain areas of the pipe into room-sized combat areas, add a little decoration based on a given theme, and call it done.

Advantage: As fast, if not faster, than any other method, without creating trivially simple maps.

Disadvantage: Often results in an unrealistic environment, and it takes some practice and patience to identify which areas of the pipeline should be bloated.

MIMICKING

The beginner's approach. Basically, steal from anything and everything you can. Copy areas and layouts out of buildings

you've been in, attach them to movie sets you've admired, and connect them altogether with an order of events that worked in your favorite game. As long as you're mixing multiple sources, you can result in some fairly decent levels while learning from the pros along the way.

Advantage: Creates much better content than you could have on your own.

Disadvantage: Makes you feel dirty and guilty inside. Do it if you're looking for a way to learn as you go, but try to wean yourself of it as soon as possible.

BLUEPRINT

The architect's approach. This actually involves sketching out the area as if it really existed – where do pipes go? Electrical wires? How and why are different areas (locker rooms, restrooms, offices, closets) placed where they are in relation to everything else?

Advantage: This method leads to the most believable environments.

Disadvantage: Slow, slow, slow. Maybe try it out for some early maps, but in the end, most of the money, energy, and time should be spent tending to the content that the player directly sees and experiences. Strive to see if you can figure out some patterns or rules to follow to simulate the same effect without putting so much invisible work into it.

AREA SKETCHING

The artist's approach. This technique is very popular in the industry, and often shows up in collector edition released concept art. Make 3D sketches of areas you'd like to fight in – from the player's perspective, or from a “security camera” $\frac{3}{4}$ angled perspective from a room's corner against the ceiling. How do obstacles fit together? From where will the enemies come at the player?

Advantage: Creates memorable areas, vastly more so than any other means of design.

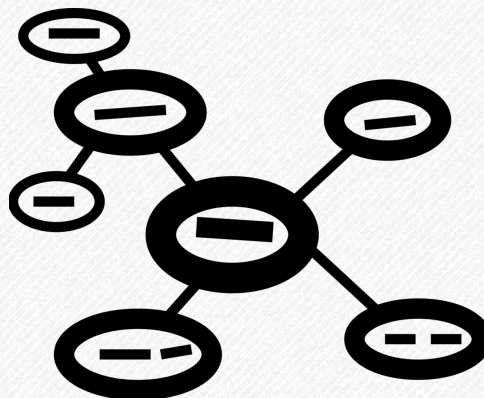
Disadvantage: Requires a little art talent, and some random inspiration. Do this when you can, but don't expect it to work for every room of every level.

MIXTURE

This is the game designer's approach. Unfortunately, it's also a lot less formulaic in how to perform it properly. Knowing some of the other prototyping methods are there to work from, and how to articulate the different aspects to team members, is essential to succeeding in hybrid level prototyping. As design schemes can be mixed, so too design prototyping methods can be mixed.

LEVEL DESIGN PROCESS

In some ways extending the previous section, here's a deeper step-by-step way to think about level design in layers to maximize output while limiting wasted work. These can apply to non-spatial stage designs, too.



Knowing how to use a paint brush is different than knowing what to do with it. Likewise, learning how to operate a level design tool is a necessary first step, but different from the more substantive task of figuring out what to do with it once learned. Let's dig into the latter.

The Process

Although every game's level or scenario design is unique, there is a common series of steps that, when followed, are likely to produce better results while

wasting less content. Without a sound process, it's all too easy for a team to work themselves into a corner, having to decide whether or not to throw out completed programming or art work because the level they were made for stinks or to have a designer floundering in poorly defined and inconsistent conceptual space from having not made the right choices prior to fiddling with the level editor.

I used these steps to make overhead 3D levels in Shotgun Debugger, I adapted a version of

this process when developing levels on the Boom Blox team, and it's the same approach that I applied to levels for Vision by Proxy: Second Edition (played more than 7 million times!).

11 Stages of Level Design

1) CONCEPT

What's this level about? What purpose is it intended to serve?

2) ARC/EVENTS

Make a short list of major inflection points. When in the level is there a change to the player's goal (next win condition), knowledge (story or motivation), or capabilities (mechanics and powers)?

3) LAYOUT SKETCH

This step is done on paper, or in a text file, and is still less about details than big picture.

Where are those arc or event points in relation to one another? What order are they seen in, and what order are they likely to be taken on? An example of a classic non-linear pattern, which has been around since the days of Pitfall 2

but still used today, has been to start the level with the end objective on-screen but somehow unreachable until later.

4) ROUGH PLAYABLE FORM

Here something is finally built in-engine, then played. It can be ugly, parts of it may not work yet, but the goal is to get enough together that it's clear how the game's mobility and interactions will affect the development and implementation of the intended ideas.

5) LIST OF ADDITIONAL ASSETS

What does not yet exist that will need to be made in order for this level to work? Early in a game's development, this can be fairly extensive, and help define the building blocks; later in development, this list should generally be kept short, in favor of reusing and recombining what has already been built.

The level may require additional programming (jet pack?), it will often require additional art (soda machines to decorate a break

room), and occasionally will pose other types of skill demands (dialog writing, puzzle planning, sound creation, etc.). Whatever will need to be done for the level to work, however minor or unrealistic (in part because that difference may be hard to know at the time), this should be acknowledged before going forward.

6) PRUNING

Taking into account the development complexity of its parts, balanced by considering the bang-for-the-buck given to the overall benefit of having those components, what can be trimmed from the level's design?

If something can be taken out without major complications, that increases focus on what is important, to both the developer(s) and the players.

7) FEEDBACK / ITERATION

If you are working on a team, this is a good point to pitch the level to others for critique. If you're working alone, now is the time to

question whether the level is good enough to finish.

A lot more work is about to go into making the stage presentable. However, if this particular attempt isn't working out so far, the types of work that follow are not going to be able to make up for that. A few steps backward may be enough, but this step occasionally leads all the way back to the drawing board, resetting to Step One. Sometimes, that's for the best.

8) FULL IMPLEMENTATION

Whatever additional programming, art, and audio are left in the level's requirements, here is the time to get to work on them.

This far into the level's development, it's clear that the list to be implemented will be supporting a level which has at least passed critical feedback and personal reflection. The place in the game for this work is known, rather than simply creating a pile of functionality or imagery then hoping for their effective use.

9) POLISH

If there is a timer or goal score, it's time to tune them. If there are enemies and items hand placed in the level, it's time to experiment with adjusting their locations and quantities for maximum effect.

10) AFFORDANCE PASS

Affordance is a word from design, used to refer to, among other things, the signals and queues on interfaces that let us know what can be clicked on, what can be dragged, and what sort of door handle should be turned or pushed or pulled.

As applied to level design, affordance means making sure that there are adequate clues for where to go next, how to progress, and so on. If the player gets stuck from being unsure how to advance, but could trivially move forward if informed of what to do, the failure is in affordance.

Having seen the level develop at every phase, and imagined its event flow before it even existed as a list, the level's designer is

often precariously blind to this sort of issue, but it's helpful to at least attempt to correct this before...

11) TESTING

...putting the game into a new person's hands. Someone that has not played the game, or has only played the game up to this point (levels 1-4 before testing level 5) can be a lifesaver in identifying the misunderstanding or confusion that may arise from failure to properly communicate affordances.

Testing can also be a helpful check on a game's difficulty, since for the same reason that the level's designer is blind to affordance issues, the level designer is probably unaware of how hard the level is now that it's tuned for someone that has practiced it hundreds of times.



Done! Name it (if applicable), finalize the writing for it (if applicable), then fit it into the flow of other finished levels. It may

need to be revisited later for consistency with other levels, or tweaked after additional testing of the full game, but all that can be done for a single level has been done.

Assuming the game is not a single level project, this next section may help.

Strategies for a Level Set

Most games come with a sequence of levels. Here are some approaches to create a series of levels that work together without blurring together:

ASSIGN A FICTIONAL FUNCTION

Give levels fictional purpose, and they'll take on unique feels.

One might take place in the palace dining rooms, where bombs are stored in underground caves, inside of an oil rig, and so on. Putting levels in those settings shouldn't just be a matter of window dressing, and sometimes building levels inspired by the constraints and features of types of locations, even without the

specific decorations to recreate them, can capture some of their quirks and uniqueness.

DISTINGUISH AREAS BY COLORS

Going from a predominantly blue set of areas to a predominantly red set of areas feels like progress. It ties together levels that belong together, and alerts the player to tune in for the other differences between the levels that don't.

Color changes are a cheap solution, in terms of time required, and whether it's literally a tint or a careful shift in an area's entire palette, it can be surprisingly effective.

Color changing is how progress was shown in Tetris. In BioShock it's why the color of the loading screen changed based on what save the area was made in, and in Alice in Bomberland it's how I differentiated later levels. In Doom the levels changed from grays and greens to reds and yellows; in Quake the episodes differed by palette from browns to blues.

DISTINGUISH AREAS BY MAIN MECHANIC

These sort of high level concept differences at an early stage of level planning can lead to drastically different level designs and experiences.

One zone might focus on destruction, another might focus on cooperation or collection. Is this area mostly vertical? Or perhaps this area leans heavily on a particular item in the user's inventory, or is built around how a particular bad guy moves?

ELEMENTAL OR BIOME SETTINGS

Lava, swamp, ice, wind, castles, ghosts, underwater, desert, moon – settings like this are cliché, but they work, and in addition to major visual differences they tend to inspire unique enemy types or player abilities.

Star Wars provides an example of applying this technique in a way that feels much less cartoony than we generally see it applied to videogames.

ONE-OFF SPECIAL CASES

The pacing of a game can be improved considerably if one or more areas of the game receives special attention. If only one area needs a time bomb count down, collapsing ceilings, and pulsating red alarm lights, then it's still probably worth doing, since the scene can have such a dramatic impact on the overall experience (Metroid, although we borrowed it for Shotgun Debugger, and Cave Story does something like it, too). When Earthworm Jim introduced stages that had glass submarines, or escorting a puppy, it helped break up the monotony of combat platforming. Call of Duty games have featured this sort of distinction in the form of waiting-for-reinforcements, tank driving, firing from helicopters, and pure-sniping areas.

PROVIDE PROGRESS CLUES

Like color changes, this is primarily an art effect, but this can help the overall sequence feel like the order matters. Something appears far off in the background,

and gradually the player sees it become closer and closer as progress is made. For a series of levels in Quake 2, there was a huge artillery gun in the skybox. In one level it was very far away, in the next it was closer, and eventually a boss fight takes place right next to it. That spatial marker helped provide a clear sense of progress between stages, even when some of the stages played very similarly. Ico employs a similar effect in how it portrays the rest of the castle during outdoor scenes.

OVERPRODUCTION

Make 25-40% more levels than are needed – stopping just short of the Full Implementation work – then throw out the worst 25-40% before going any further on them. This allows a filter of editorial judgment, makes it possible to stumble upon interesting ideas without misusing the implementation time of others, and encourages a degree of exploratory risk taking that's

absent when everything started is expected to be used. If the second level in this batch doesn't fit in with the others, but seems like an idea worth keeping, it may inspire a other set to be produced then pruned.

This approach leaves level order a bit more up in the air, which sometimes can only be known after the levels are playable in-engine. If the third or fifth level made in a given batch seems easier than the first, or seem to better introduce concepts assumed by the second, it's not too late to rearrange them then account for that (in terms of polish, ordering of art or enemy types, etc.) when moving into the later stages of implementation.

LEVEL DESIGN Q & A

Here I provide more details and explanation for some of the ideas just mentioned in Level Design Process. It also emphasizes that developing levels often begins with an exploratory building phase for practice and research.



Hi Chris. As part of my studies, I'm required to undertake a research project. I chose level design for my topic. I recently read your article on Level Design Process and found many of the steps and strategies quite helpful. I was wondering if you could be so kind as to spend a few minutes to answer some other questions.

1) DO YOU START WITH THE END IN MIND, OR SPEND TIME BRAINSTORMING AND PLANNING THE LEVEL?

I don't do either, at first. I begin by doing plenty of tinkering and exploratory building, to ensure that I'm working with the engine to do what it does best. Exploratory building involves making each

attempt as different as possible from my other attempts, so rather than iterating rigorously trying to achieve a predetermined concept, with each attempt I'm switching up my methods, priorities, emphasis, and so on.

When something in my exploratory building seems to be working out better than the other attempts, say for example it's the best out of four or six attempts that I've roughed out, I'll try to figure out what's working well about it then extend it to amplify why it's

working. The reason might be conceptual, or emotional, like playing up a sense of power or panic, but often it's something more concrete like how a specific enemy, weapon, item, or feature plays well in some circumstance. This can also arise from discovering implementation quirks; if there's a special gimmick I've discovered in the game's engine and level format that's not too buried and not obviously just a bug needing to be fixed, I may make a level around introducing and exploiting that.

Sometimes that's just enough material for one level, but in other cases it can become a source for a number of levels, each presenting some angle or variation on whatever worked.

You might wonder, how do I identify what's "working"? There's no science to it, but roughly speaking I do it one of three ways:

The first way is through exploration and reflection.

If it's one of the first levels I'm making for the game, the tools are often still being figured out. There's not much to go on besides a personal matter of taste. Taste isn't always a good guide, and there's a reasonable chance these will later either get cut or updated.

The main idea driving early building is to learn more about the game that you're working on. Which items, weapons, or enemies need to be used sparingly, versus which can I get away with using quite frequently? Are there patterns I'm discovering that can be tucked away mentally to apply elsewhere, such as for example some puzzle-like arrangement or particular type of ambush set up? I evaluate those levels based on a variety of criteria, like how unique they are, how fair they seem, how they look visually, and so on, then work out some simple way to rank them based on those scores and try to learn from that. In this way, I ground my work in the nuances of

how the engine and implementation interplay with the player.

The second way is applying what I've learned.

With a base of existing levels to then consider, at this point I may do some rough sketching of map shapes. This is closer to the beginning with the end in mind, though I still largely let the level design itself a bit during iteration. I won't scrutinize over the details on paper, which I'll instead sort out while working in the editor and format (in other words: expect to be making changes during translation to playable level), but beginning on paper makes it easier to explore different overall shapes, structures, and set ups than what I would produce just fiddling around in the editor environment. Since the first phase was partly just about fooling around in the level environment, if I kept doing that for this second phase I'd quickly start repeating myself.

The third way is just by filling in gaps, and fixing maps.

When the game is nearing completion, I'm on the lookout for ways to fill in information unaccounted for in the existing levels. This can be a bit less exploratory, since here I'm trying to identify and address specific issues. Are more easy stages needed, to better introduce some of the elements that other stages use more abusively? Is there an appropriately hard and rewarding stage to bring together skills and knowledge that the player has gained, for later in the game?

If there's something that seems to lose the player by coming out of nowhere while playing an existing level, I might try to refactor that part just before it, so that it can come out more smoothly in the transition to that idea. If I can't seem to get that right, rather than forcing it, I might just cut whatever's causing the problem.

I typically build the later levels (middle to end) of the game before working on the early levels that build up to them. This way, I have a clearer idea of specifically what the player needs to be prepared for in order to be ready for the main levels, based on what worked for those fleshed out challenges. Otherwise starting with the game's initial levels would have to provide an unfocused introduction and practice to ideas that may or may not be relevant to what works well for later stages.

Within the development of each level, I tend to work in iterative passes, with a different goal at each step of the process. That process is outlined in the level design entry you already read though, so I'll avoid repeating it here.

2) HOW DID YOU COME TO LEARN THE SKILLS YOU NEED TO BE ABLE TO CREATE EFFECTIVE LEVELS?

Whatever I know about level design comes primarily through practice, in particular working on different kinds of levels, working

with different types of tools and processes, and designing levels for different genres of games. Vision by Proxy Second Edition, TriChromatic, Swarm, Battleship 88, Shotgun Debugger, Boom Blox, Topples, Zylatov Sisters, Exxak, Freezing Solid, Crystal Ball, and Vectorverse all have completely different level formats, gameplay styles, and tools.

I will qualify however that designing first-person shooter levels, or for that matter third-person adventure game levels, on account of their architectural complexity and sheer volume of detailed artwork, have their own specific practices different than the ones I'm describing here. The types of levels that I do most of my design for have either comparatively simple architecture—in some cases no physical structure—which means I can generate new maps very quickly. Although I didn't do any level design that wound up in Medal of Honor Airborne, in part because I

was finishing school during that stage of development, earlier in that project I had an opportunity to work on prototype levels with members of the team, and the process overhead is much higher for spaces that large and complex that need many people with different skills to always be on the same page.

3) HOW DO YOU GO ABOUT PLANNING THE OBJECT PLACEMENT AND INCLUSION IN THE LEVELS?

I start by coming up with vague conceptual plans for areas that seem appropriate for their placement and structure (deciding for example which area I might want crowded, safe, a climax, etc.), which I then translate into a playable form by scattering about an initial arrangement of enemies and items, puzzle elements, whatever. After that it's largely a process of iteration to improve what's there, combined with cutting away any sections or ideas (say, the worst 25-60%, or sometimes a whole level if it just

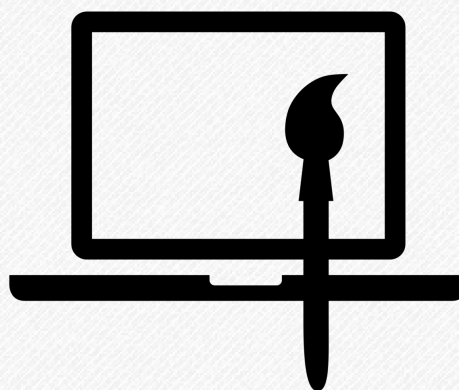
isn't working out) that aren't shaping up as well.

4) WHAT ADVICE WOULD YOU GIVE AN AMATEUR LEVEL DESIGNER WISHING TO PURSUE A CAREER IN THIS FIELD?

Make levels! Make levels for a variety of types of games, for a variety of different purposes, and don't feel stuck to using the tools that you already know (be willing to learn new software, techniques, processes, etc.). Ideally, not as a short-term goal but as a target, try to create something that strangers will go out of their way to play and then tell other strangers about. That may seem obvious, but with so many options out there for videogames at this point, it can be surprisingly challenging to earn a stranger's time and attention, and certainly their recommendation. That's a good test though of whether what you're doing makes sense without you present to explain and promote it. Because at some point, your content needs to explain and promote itself.

NON-ESSENTIAL LEVEL ART IS ESSENTIAL

A level is not just a matter for game design - it overlaps substantially with art considerations too. The layout has to be pleasing. It has to not be too sparse nor crowded. As part of that, it also needs to be decorated!



One of the differences that can cause a student or novice project to stand out in a bad way, looking and feeling unfinished, is the lack of level art that's non-essential to gameplay. I'm referring of course to art other than the main background, only because I'm assuming that main background art's already accounted for.

Such non-gameplay art can be divided into three categories:

VISUAL-ONLY

This kind of art has no collision data and is pure decoration. This

might include level pieces that actually do handle collision but are outside the playable area (though be careful to not confuse the player into thinking non-reachable areas are somehow reachable). This provides visual interest, and can be more flexible for landmarking areas because the density and useable configurations of collidable level pieces is limited by their consequences on player navigation and action.

COLLISION-BUT-DECORATIVE

This includes minor obstacle, potentially adding a bit of variation to enemy movement and line of sight, but mostly just something to navigate around, or something to jump over. This provides a reason to pay attention to the player's character and input while walking from point A to point B, instead of simply holding a button down for screens at a time.

ALTERNATIVE-MAIN-COLLISION

If an overhead level's shape is defined by placement of trees, add boulders into the mix. If there's water – even if it isn't traversable – have two shades to distinguish shallow versus deep. If a side view level's main platforming tiles are metal, also have a set of wood; or if you're feeling lazy, tint that same metal blueish for steel or brownish for rusty metal. In addition to adding a bit of visual interest, changing usage ratios or switching between sets of these helps convey a sense of meaningful progress between noticeably distinct areas.

“I've made it underground to the caves,” or “I'm finally outside the castle walls” is just a tile swap, but without that tile swap, it's merely walking between rooms that are otherwise pretty indistinguishable.

There doesn't need to be a ton of non-essential art to make a major improvement in a player's perception of the game. You can get a lot of mileage out of even a handful of relatively easily created assets. Having a little bit of non-essential art, especially if used in all three of the areas outlined above, can improve the first impression, likelihood of returning to the game, and memory about the game later.

Although I'm calling it non-essential, I mean only that, strictly speaking, it's non-essential for gameplay, i.e. that the game can be feature complete and fully functional without it. But make no mistake: this kind of art is essential for the game to not feel barren, for the game's levels to not

all feel the same, and for the game to look done.

There's really no excuse to avoid doing it. By design, player attention isn't focused directly on these uses of art, so repetition is pretty easily forgiven. Nor is it especially time consuming compared to most other types of game art, since it's rarely (or only very slightly) animated. And, again, players don't typically look directly at it, so quality control and iteration isn't really as important as it is with art for power-ups, enemies, or player sprites.

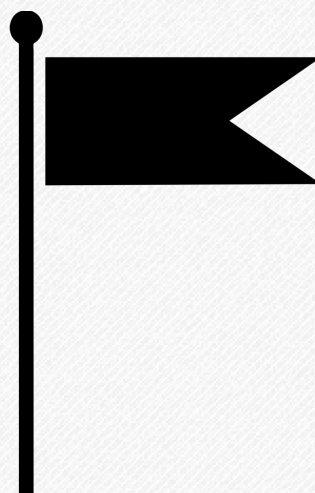
For games with more involved level design, placing these decorative elements into stages can often be put off until later in development. This way stage blocking and navigation can be figured out with basic walls. This is helpful since we'll often completely discard some layouts that aren't working out as well before we invest additional time decorating something that doesn't belong in the final version anyhow.

These types of art assets are the icing on the cake.

And a cake without icing isn't ready to show anyone.

VISUAL LANGUAGE IN SUPER MARIO BROS LEVEL ENDINGS

Classic games, especially those that helped shape the industry and create a place now held by digital worlds in mainstream culture, are often rich with design insights just waiting to be mined and put to new uses.



Every level of Super Mario Bros (1985) on the Nintendo Entertainment System ended with either a tall flagpole or a battle with Bowser over a bridge.

Super Mario Bros has 32 levels – 8 worlds, each with 4 areas. The 4th area is always a castle that ends by getting past Bowser.

FLAGPOLE SCORING

On all flag levels, the player gets more points for touching the pole higher when crossing. 5,000 points are awarded for the very top, 2,000 for even a little below

that, awarding only 100 for crossing at the very bottom. To help the player reach the top, nearly all levels end with a stair step structure. Additionally, these structures (with only a couple of exceptions) are built to disallow second chances. The areas were shaped to prevent Mario from climbing back up after a failed jump. If air control is used to back away from the flag mid-jump, in these cases the best the player can do is 800 points (running jump), while a standing jump will



Level endings: the top left is Level 1-1, the top right is Level 1-4, the bottom left is Level 8-1, and the bottom right is Level 8-4. Note that all water and dungeon levels end above ground, in a common section.

only earn 400, since Mario jumps higher when running at full speed.

VISUAL LANGUAGE AND EXPECTATIONS

The function of these stairs, in connection to the flagpole scoring, creates a visual language that the player learns to interpret. When the player sees steps coming on screen, rising from the ground to

the top (Super Mario Bros does not scroll vertically), it's a hint from the level designer that the end of the stage is near.

By establishing this visual language, the player can also be teased a bit. When the level includes a partial stair pattern mid-level, that implies the the

stage may almost be over. Taking just a few more steps exposes a pit or some other violation of the pattern, letting hopes back down, but as shown in Levels 5-1, 5-2, and 5-3, those inconsistencies become mixed into the end steps, complicating the player's ability to rely on knowledge of the end pattern. That complication translates to more false positives and false negatives in the remainder of the game.

This pattern also instills an expectation of safety. Since most end stairs have no enemies, it's even more dangerous when that pattern is broken by enemies placed in the final step area.

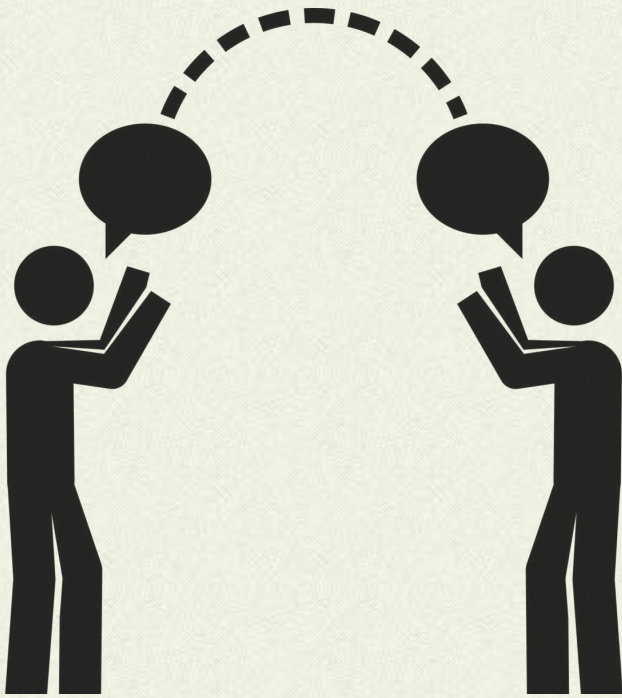
Notice too the misleading clues during play about which castle will be the real, final one. Level 6-3 has a uniquely colored castle, presented at the end of the second night world... after which still another daytime level is presented. The game then breaks the 2-day, 1-night pattern, ending

on a second day chapter in a row without the night to follow.

Nothing in the game reveals the player's overall progress toward completion, either as a percentage or by exposing the total number of worlds in the game. Unlike a book, it's impossible to size up how much of the game's content remains, and unlike a film or TV show there was (and still is) no convention for a videogame's length. This suggested that no matter which area the player loses in, practicing to make it a tiny bit further might complete the game.

This deliberate uncertainty creates more excitement about reaching a new castle – due to thinking it might really be the last one – and added genuine surprise to seeing Toad each time in place of Princess Toadstool. Until, of course, the player finally finds Princess Toadstool, by which time the player has been conditioned to expect to see Toad – making the reward for finishing one final surprise.

7

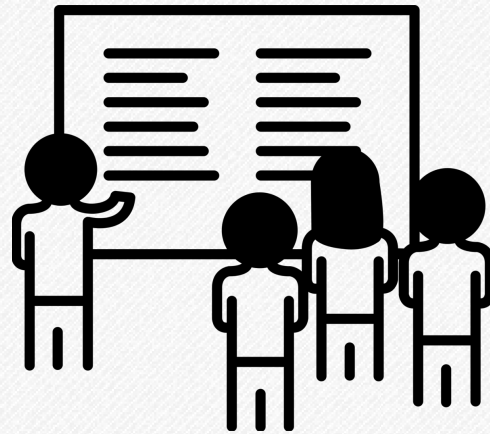


Being able to work alone has its advantages. No one to answer to. No one to compromise with! However learning to work with others is a skill. Once mastered, it opens up new opportunities to collaborate with others, making it possible to involve those skills that others devote their lives to.

TEAM PROJECTS

VIDEOGAME PROJECT MANAGEMENT FOR HOBBYISTS AND STUDENTS

All managing is difficult. But managing without having an official role with authority brings added challenges! Here are some practical tips to simplify matters a little, enabling your team to focus all attention on the game.



Joe L. wrote to me:

“I’m confident with my ability in programming and art. One area I’m less confident in is my ability to mentally visualize a process from start to finish. The act of mapping out a project can be very daunting to me. Any advice you can offer on good game project management would be extremely helpful. Tips on avoiding quicksand would be great.”

As he’s rightly pointed out, videogame development skills like programming, design, and art don’t transfer clearly to project management, nor is it natural or

automatic. Project management is a fundamentally different skill to be studied and practiced.

When working on our own, the stakes are lower. The risk of wasting time by being overly ambitious puts no one’s time at stake but our own. Uncertainty about what’s happening next isn’t blocking someone else’s work. Communication of our constantly changing ideas is a non-issue.

As soon as the time, talent, and future prospects of others are

involved, the stakes go way up. Even at a student or hobbyist level, not knowing what you're doing at the helm switches from exploratory wandering (when done alone) to high pressure responsibility (when done with the time of others).

Even though a lot of these points may also relate to solo project planning, my focus here is on team projects, since that's when this sort of work becomes particularly important. Even though I suspect some of these relate to project management in general, including how it may be relevant to larger commercial production, my experience in those areas is overshadowed by my time on student and hobbyist projects, and so I'll leave any such extrapolations up to reader discretion.

While leading my first project teams, I hit a number of these discoveries through trial-and-error. When I began helping with more team projects that were run by

others, I picked up that these challenges or methods were by no means unique to my own particular background or style. Here are some things that I wish I had known sooner.

MOCK-UP SCREENSHOT AS DESIGN DOC

Photoshopping a "screenshot" (i.e. not really a screenshot) before the game exists goes a long way. A per-pixel representation of what a moment of the finished game might look like – with consideration given to the sort of graphics work and programming that can fit into the time available – concisely communicates dimensions, layout, and user interface elements. The programming focus becomes an exercise in bringing that screenshot to life, which prioritizes gameplay first (as opposed to menus/etc.), and offers clear reminders of what's missing (health bar serves as a reminder that health and game over need to be hooked up, etc.). It doesn't

need to be final art quality, but it should be presentable.

In the process you'll give artists a unified sense of scale and dimensions, level designers an idea of the structures involved, and dozens of questions that no one would have thought to ask explicitly are answered simply by putting the picture together.

ONE SIDE OF ONE PAGE DESIGN DOCUMENT

In terms of design document, if there is one, don't let writing or maintaining the document take over or cut into development time.

If it can all be on one side of one page, briefly explaining what happens and what's involved, that will have more utility than a 10-20 page document that no one reads and has too many interconnected assumptions to be easily updated.

FOCUS ON ONE OR TWO INNOVATIONS

Unless the project's purpose is to be an experimental art game: don't be afraid of applying some control, interface, and design conventions. Experiment with, at most, one or two aspects per

project, and for the rest focus on execution rather than the high level questions. It's okay for the health to be a percentage bar or a row of hearts. It's okay to collect items to earn extra lives. Leonardo da Vinci didn't reinvent paint, and he didn't invent the woman, either; he changed the way that billions of people have seen the world since by simply having done a better job of painting a woman.

TAPPING OTHER GAME REFERENCES

Referencing games that others know on the team can be a powerful shortcut. If a platformer is being worked on, there's a broad history of existing platformers that could be referenced to clarify what sort of jumping, attacking, or movement is planned. Particularly with the explosion of free-to-play, work-on-any-computer Flash games, there's almost always a few handy references that can be linked to for general gameplay or interaction demonstration. It's good to not rely entirely on such references,

though, since inevitably some otherwise interested people won't know the games mentioned, and need to get the gist before they'll investigate examples.

If there's no example out there demonstrating what you have in mind, you need to make one first (see next point).

Note that the clarity that comes from this sort of communication can be a double-edged sword: suggesting the game with a five month development window is going to be "like Halo," or even better "like World of Warcraft", reveals how little a would-be producer understands about game development, which will keep serious developers from joining the march over the cliff.

BUILD A PROTOTYPE IN ADVANCE

If it's possible to get a playable prototype (the core of the game, ready to be expanded upon) working prior to bringing others onboard, this can help tremendously in ensuring people are on the same page, and

interested in a compatible direction.

If nothing exists that can be pointed to and clearly described as, "like that," then the prototype fills the role of making that communication possible. Even if an outside example already exists (Mega Man? Space Invaders? King's Quest?), putting together the core gameplay before bringing others onboard is proof that you're capable of putting a project together. "I know how to program it" doesn't win much credibility; "player movement and platforms work already, we're ready to focus on animations, sound, power-ups, enemy movement, and level design" shows that the project is ready for that kind of work.

This is about addressing a potential time black hole before it's too late. The less has been figured out when reaching out to others, the more disparity there can be in expectations – some people coming onboard imagine it as a 3/4 overhead action game,

some imagine it as a first person adventure, some imagine it as an RPG – and that mismatch in interests can dovetail the project throughout as members pull it in different directions to be more like what they want to work on. “It could be about sci-fi, wild west, modern day, or abstract visuals” may attract people with 4 different types of interests, many of whom may either be saddened by the direction it takes, choose to leave when they don’t get their way, or thrash until the last week of production trying to wrestle back to what they joined for.

By contrast, if the core of the gameplay and concept is stood up beforehand, so that everyone can see and try it, people that have interests aligning with the direction it’s moving it are more likely to self-select.

CLEAR FUNCTIONAL ROLES

Clearly define roles to make accountability clear. One person can of course wear multiple hats, provided that they keep up with

those respective duties. If roles are ill defined, it’s likely to run into a variety of stepping on each other’s toes and diffusion of responsibility, both of which cause time to be lost to thrown away assets and hopeless confusion.

TRACKING ISN’T THE SAME AS LEADING

Facilitate, motivate, and make sure everyone has as much to do as they’d like to be doing – it isn’t sufficient to track what people do, any more than it’s a football coach’s duty to simply record what happened after each play, or a general’s to jot down how things unfolded after the war. Being aware of what everyone has done in the past week is important, but it’s equally important to be aware of what everyone is doing the following week.

People like clarity. People like the confidence of direction. People like to feel like they’re a part of a coordinated effort.

A word of caution: note that this is not about giving orders, bossing others, or stamping out ideas.

Rather, it's about having a plan. That plan can (and often should) be tentative and up for discussion, but having one helps ensure that if everyone just chugs forward roughly following the shared schedule, the work will all come together in a meaningful and worthwhile way.

WELL-RUN MEETINGS AREN'T USELESS

People complain about useless meetings. They should. Useless meetings are infuriating. Useful meetings, by comparison, are terrific. Meetings aren't the problem – poorly run meetings are the problem.

Run correctly, meetings are efficient, get everyone on the same page, clarify objectives for the week, reassure those involved, and result in some discoveries or action items that would have otherwise gone unnoticed.

There are things that come up in person which absolutely won't come up by e-mail or text chat. Sometimes it's in the tone someone uses, sometimes people

just feel more comfortable bringing things up face-to-face, and sometimes preparing for the meeting is the only hour that week someone spends reflecting on their involvement with the project.

Have a plan for meetings. A half page list of bullet points is fine – the agenda need not serve as a standalone form of notes. This makes it more clear when various topics are coming up, increases the chance of the meeting having value, and generally is part of respecting the time of peer developers. Are there decisions that need to be made? How does the latest version of everyone's combined efforts look, play, and sound? What is there to look forward to coming together over the next week?

Table one-on-one discussions about particular aspects to be resumed at another time or setting, even if only after the group meeting adjourns. If the sprite artist and the audio person need to discuss their work to make sure

the appearances fit the sound effects being made, there's no reason for the programmer or level designer to be there unless they want to be.

Let the team members have input on project design and direction (both of which are good), but don't confuse that with counting on the team to run the meetings (which is bad). Several people trying to do the work of one is inefficient and ineffective – keeping order in a meeting is a one person job.

CLEAR WEEKLY GOALS

With accountability roles well defined, a rough weekly schedule can be drawn up – each week, what is being done for audio? Functionality? Level content?

“What are you doing this week” should never be answered with, “More programming.” The goal is not to spend a certain amount of time every week in front of a computer typing – it's to get more things working, or to fix things that are broken.

Better answers are along the lines of “By Friday, we'll have the level format saving and loading,” or “By Friday, we'll have the second half of our sound effects hooked up with placeholder files.” The weekly plan can be adjusted, if the schedule begins to fall behind, but it's better to do that as soon as reality is starting to take its toll, and not on the tail end of a project after other roles drove forward under the assumption that certain things (3D animations, boss fights, whatever) would be ready.

Who will be doing what each week? What is one tangible result that each team member can independently take on during that time? Are there things the programmer or designer can do in a different order to unblock another's animations, audio, or other work showing up in-game sooner?

SCHEDULE BACKWARD FROM COMPLETION

Creating each week's goals one week at a time is the equivalent of driving around the highways until

we run out of gas, hoping that when the car stops, we're happy with where we are. That's insane, but it's also a surprisingly common way for first-time project leads to manage a team (regardless of how much game development they've done on their own).

An excel spreadsheet can be a lifesaver for this – early on, make each column a Friday date, make each row a developer's role, then do your best to figure out how to fill in those remaining cells for who might be able to realistically get what done each week. Work backward from the end result, or if you've already started some aspects of production, work in from both sides and meet in the middle. Work with team members to figure out if they see their involvement as different, expect that they'll have more or less time to offer, or have suggestions on how to more effectively use the time available.

That spreadsheet is also a handy tool for peeking at weekly during

development, to see what's falling behind and needs to be re-evaluated. If any developer's objectives start to fall two weeks or more behind, and strategizing with that team member doesn't seem like it's going to put things back on track, find a way to design around the loss of that work, having less of what that developer projected, or picking up the slack (either yourself or via another reliable team member).

TIME ESTIMATING: MIN/MAX/AVG

Figuring out how long something will take to do is a notoriously difficult thing to do, especially if it isn't something that you've done before. If it's not your skill area, be sure to speak with someone that does it when working out the initial schedule. Even then, people can have a hard time, since there's inevitably some flexibility in how much time gets poured into each thing. One simple technique around this is to ask for how fast it could possibly be done (minimum), how long it might take

in the worst case (maximum), and then base estimates on roughly the average, tending toward the maximum end.

There's a tendency to think in best cases only when scheduling, which runs into disaster when at least half the time conditions are far from best case.

1-2 NIGHTS PER WEEK PER DEVELOPER

For students and hobbyists there's coursework or rent-paying work to contend with, and so estimating for 1-2 focused nights of effort per week per developer tends to be fairly realistic. If someone can fit in more than that, great (though I would advise against expecting it to be consistent, as competing deadlines or life issues inevitably crop up). If someone can't fit in 1-2 focused nights of effort per week, then they aren't able to put enough time in to make a tangible difference on the project anyhow, and may as well not be with the project.

There are, of course, exceptions. Certain specializations, like music

composition, manual writing, or menu design may only be relevant for a couple of weeks in the project, depending upon how someone prefers to work, and what else has to be done before they can start (it's often hard to put together a cohesive manual PDF or readme until the game is almost done). In those cases it's of course unnecessary to expect that they hang around doing something weekly.

There's generally no reason to have people track their hours, or to be Draconian about someone missing one week due to a perfect storm of exams/deadlines/personal. The 1-2 night estimate is simply a rough guideline for modelers, animators, level designers, etc. to put a meaningful dent in the project week by week. Not 5-7 days per week (unrealistic), and not 0-3 hours per week (not useful). The important thing is that the developer's objectives are getting done, within a week or so of when estimated,

and adjusting plans if necessary if it turns out that the original schedule estimates were overly optimistic.

DIVIDING WORK REQUIRES MORE WORK

Involving multiple programmers takes a substantially different type of programming expertise to structure shared code properly (or else a very clear cut separation). Having 5 programmers doesn't get the code done 5X faster, it adds a substantial overhead of communication, misunderstanding, necessity for internal documentation, and disagreement about abstract details that (mostly) won't affect the outcome.

If art is split, it should be carved into distinct compartments that won't result in inconsistent visuals serving in parallel roles (ex. one does background art, one does characters, one does power-ups). This also minimizes communication overhead, and the likelihood of running into a choice

between rework or bizarre inconsistencies.

If there's one member solely responsible for the audio, it's less likely to have dramatic variation in volume and duration. Music composition, of course, separates nicely from sound effect production, and voice recording (if present) may also be split out if necessary.

In general: if it can be done well by one person, don't have it spread between two.

FOCUS DESIGN TO FIT TIME AVAILABLE

A lot of developers do this backwards, extending their schedule arbitrarily to keep up with expanding design ideas. Those projects, even the ones with millions of dollars, a full staff of experienced professionals, and years of development time tend to Duke Nukem Forever into the abyss of unfinished games.

If people join the team with the expectation that the project will take 5 months of commitment, as

the producer, you are the guardian to ensure that promise is kept, and that there's something complete to show at the end of that time period. If others want to stick around after that initial release and work on an expansion pack or sequel together, more power to them, but don't let others in the group be held hostage by a poorly managed schedule dragging on indefinitely before they can share the fruits of their labor.

(Note that, fairly reliably, the project's production will tend to slip a little beyond the time window, as a matter of tying up and fixing things leading to more tying up and fixing things. That's true even if that inevitability is accounted for in planning - i.e. Hofstadter's Law: *"It always takes longer than you expect, even when you take into account Hofstadter's Law."* Don't be surprised when it happens, try to not to use it as an excuse, and tough it out.)

SCOPE CONTROL

About 3/4 through the project's allocated time, make a list of what must be done for the game to be presentably finished – not necessarily complete to the original vision, but first-and-foremost done. Only allow that list to get shorter, ideally by cutting anything on that list if it turns out to not be essential.

Rather than seeing great disparity between what was originally in mind vs what's in the game, try to think in terms of starting on something "new" with a substantial head start. Imagine you're given a pile of assets (conveniently, the art, sounds, levels, and functionality currently yielded by development to date), with the challenge to weave it together coherently into a finished videogame, which may or may not be aligned with what was originally envisioned.

When features and requests come up mid-development that seem tempting, but would either take

the project in a wildly different direction or add significantly to the game's development needs (say, adding RPG-like character dialog to what was previously a side-scrolling action or overhead strategy game), table the suggestion for a sequel. If the core game initially released turns out great, those team members passionate about those additional ideas have a strong foundation to build upon. The initial release may even be helpful in pulling additional talent onto the project more relevant to new types of challenges, filling in gaps created by others that choose to move on to something different.

PROTECT THE TEAM MEMBER'S TIME

If there are five people involved in a game, all of whom came onboard agreeing to an initial vision, and then one of them suggests a change that will take the project in an entirely different direction and grossly expand the time required, don't feel bad about being resistant to that request.

Instead, feel good about defending the time and work of the four other people involved. That's a different game, to be taken on a different time, and if they wish to assemble a team to build that vision, nothing prevents them from doing so separately. (Of course, if the idea is lobbied effectively to others, with half the team sold on it, that's another situation entirely. The best course of action in this situation is significantly more blurry.)

GOOD ONES NEED YOU TO 'FIRE' BAD ONES

If someone is having more of a negative impact on the project than a positive one, or even if they're simply having no impact, let them go. It's not necessary to be a jerk about it, it should never be done in front of other people, and it's good to try to discuss and attempt other options first (would they be better able to help the project in a different role?), but on occasion it's necessary for the good of the team and the good of the project.

Someone should not wind up in the credits simply because they came around, offered to do something, then were hard to get a hold of or never contributed anything. That's unfair to the people building the videogame.

Healthy perspectives on this difficult matter:

1. The person being let go usually isn't surprised – they know better than anyone how much they have or haven't been contributing.
2. As a student or hobbyist, you aren't disrupting their income, throwing off their rent payments, or turning their life upside down. If they weren't constructively involved anyway, they have nothing to lose but the idea that this was filling a slot in their life that would be better filled by something that's a better fit.
3. If they weren't using the time to work on this project, then they have plenty of opportunity to be using that time on something else instead.

4. Some people have it so ingrained into their worldview that quitting is a bad thing that they may be hoping you'll let them go, but lack the courage or conviction to quit on their own. Set them free. It may be a relief.

5. Do it to defend the efforts and integrity of others on the team, because as the producer or project lead, you're the only person in a position to do it. Nobody wants to organize a group "vote off the island" of a stray individual, because it's necessarily indicative of group condemnation and conspiring behind someone's back. Take responsibility. If three other team members are annoyed by the fourth team member seriously in need to be let go, they'll be as frustrated by your ineffectualness or lack of awareness as team lead as they are about the team member needing to be let go. It's rare, it's important, but as project lead, people are counting on you to

know when and how to make that call.

PREPARE FOR DISASTER: MAKE BACKUPS

Nightly, or at the very least weekly, make regular backups of source code, asset files, documentation, anything that the team would be devastated to lose. SVN everything to a remote server, upload a zip to your student web space, use an external hard drive – whatever method works for your budget and comfort. If a hard drive fails, whether it's yours or anyone else's on the team, that should not be the end of the project. The producer and lead is accountable for seeing to the completion of the project, and any potential risk that could lead to non-completion is up to that position to mitigate.

FINISH IT

Less is learned from an unfinished project. The thoughts are never completed, the scope is never fully understood, the project is never presentably put before others in a way that their reactions or

feedback can be observed in the context of the work that was done.

I hear “almost done” about a lot of hobby videogame projects. This is as distressing as it is inexcusable. If it's “almost” done, then do the last 6 hours already and make it done. Nobody wants to drive a car that's “almost complete,” or sleep under a roof that's “almost repaired.” Even if the game isn't a high profile project likely to get a lot of publicity, it's likely to get orders of magnitude more players if it's done than if it's almost done.

If the project needs to end earlier than anticipated, due to a team split, a dramatic schedule disruption (core people moving away usually falls into this category), or other factors, triage it. At least stitch back up the openings, however far it may be from the ideal or original vision, so that it's able to stand on its own.

We get better at what we practice. Practicing doing unfinished work

will only make someone better at doing unfinished work.

appealing excuse to slip past an end-of-semester deadline.

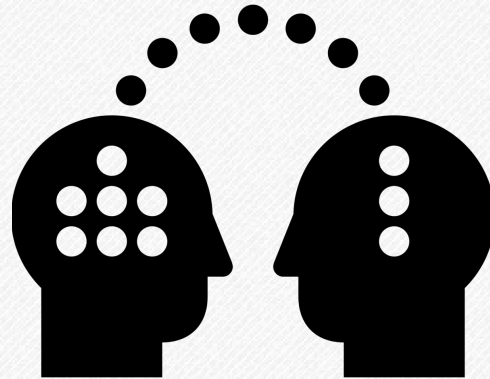
LAUNCH PLAN

After the game is done, don't release it immediately. If you can't wait, it's not the end of the world to make it available as soon as it's presentable. However, an extra couple of weeks to put together some public facing material – compelling screenshots, short description text, simple web site (ideally on its own domain), and a marketing video highlighting key features and qualities can go a long way in getting more people playing.

I'd suggest not including that launch plan in the original schedule. Keep a marketing timeline (if any) separate from the development schedule, or it's likely – as the QA and testing time at the end often does – to become subsumed as the project goes past its deadline. “Well, we were counting on not releasing until July anyway...” is a dangerously

COMMUNICATION IS A GAME DEVELOPMENT SKILL, PART 1

Many people that invest their time in mastering their craft - programming, design, art, or otherwise - find difficulty in working with others. Let's look at why this is worth fixing, and two main patterns of issues.



This subject is an incredibly important one that gets a lot less attention than it should. Most videogame development guides and tutorials fail to acknowledge that communication is a game development skill.

Knowing how to speak and write effectively is right up there alongside knowing how to program, use Photoshop, follow design process, build levels, compose music, or prepare sounds. As with any of those skills there are techniques to learn,

concepts to master, and practice to be done before someone's prepared to bring communication as a valuable skill to a team.

Game Developers Especially

Surely, everyone in the world needs to communicate, and could benefit from doing it better. Why pick out game developers in particular?

I don't meant to claim that only game developers have communications issues. But after spending much of the past ten

years around hundreds of computer science students, indie developers, and professional software engineers, I can say that there are particular patterns to the types of communication issues most common among the game developers that I've met. This is also an issue of particular interest to us because it's not just a matter of making the day go smoother; our ability to communicate well has a real impact on the level of work that we're able to accomplish, collaboratively and even independently. Game developers often get excited about our work, for good reason, but whether a handful of desirable features don't make it in because of technical limitations or because of communication limitations, either way the game suffers for it the same.

Whose Job is It?

If programmers program, designers design, artists make art, and audio specialists make audio,

is there a communication role in the same way?

There absolutely is. There are several, even.

THE PRODUCER

Even though on small hobby or student teams this is often wrapped into one of the other roles, the producer focuses on communication between team members, and between team members and the outside world. Sometimes this work gets misunderstood as just scheduling, but for that schedule to get planned and adjusted sensibly requires a great deal of conversations and emails, followed by ongoing communications to keep everyone on the same page and on track.

THE DESIGNER(S)

One way to think about the designer's role in game development is to communicate with the player through the game. Indicating what's the goal, what will cause harm or benefit, where the player should or shouldn't try

to go next, expressing the right amount of internal state information – these are matters of a game’s design more so than its programming. Depending on a game team’s skill makeup, in some cases the designer’s only direct work with the game is in level layouts or value tuning, making it even more critical that within the team a designer can communicate well with programmers, artists, and others on the team when and where the work intersects. On a small team when the person mostly responsible for the design is also filling one or more other roles (often the programming) communication then becomes integral to keeping others involved in how the game takes shape.

THE LEADS

On a team large enough to have leads, which is common for a professional team, the Lead Programmer, Lead Designer, or Lead Artist also have to bring top notch communication skills to the

table. Those people aren’t necessary the lead on account of being the best programmer, designer, or artist – though of course they do need to be skilled. They’re in that position because they can also lead others effectively, which involves a ton of communication in all directions: to the people they lead, from the people they lead, even mediating communications between people they lead or the people they lead and others.

Some of the most talented programmers, designers, artists and composers that I’ve met have been quiet people. This isn’t an arbitrary personality difference though. In practice it limits their work – when they don’t speak up with their input it can cost their game, team, or company.

THE WRITER

Not every game genre involves a writer, but for those that do, communication becomes even more important. Similar to the designer that isn’t also helping as

a programmer, a team's writer typically isn't directly creating much of the content or functionality, aside perhaps from actual dialog or other in-game and interstitial text. It's not enough to write some things down and call it a day – the writer and content creators need to be in frequent communication to ensure that satisfactory compromises can be found between implementation realities and the world as ideally envisioned.

NON-DEVELOPMENT ROLES

And all that's only thinking about the internal communications on a team during development.

Learning how to communicate better with testers, players, or if you've got a commercial project, with your customers and potential new hires (even ignoring investors and finance professionals), is a whole other world of challenges that at a large enough scale get dealt with by separate HCI (Human-Computer Interaction) specialists, marketing experts, PR

(public relations) people and HR (Human Resources) employees. If you're a hobby, student, solo, or indie developer, you've got to wear all of these hats, too!

There are two main varieties of communication issues that we tend to encounter. Although they may seem like polar opposites, in reality they're a lot closer than they appear. In certain circumstances one can even evolve from the other.

Challenge 1: Shyness

The first of these issues is that some of us can be a little too shy. As I mentioned, Some of the most talented programmers, designers, artists and composers that I've met have been quiet people. In practice it limits their work.

It's very easy to rationalize shyness. After all, maybe the reason a talented, quiet person was able to develop their talent is because they've made an effort to stay out of what they perceive as bickering. Unfortunately this line of

thinking is unproductive in helping them and the team benefit more from what they know.

Conversation between team members serves a real function in the game's development, and if it's going to affect what gets made and how it can't be dismissed as just banter. Sometimes work needs to get done in 3D Studio Max, and sometimes it needs to get done around a table.

Another factor I've found underlying shyness is that a person's awareness of what's great in their field can leave their self-confidence with a ding, since they can always see how much improvement their work still needs just to meet their own high expectations.

It doesn't matter though where an individual stands in the whole world of people within their discipline, all that matters is that developers on the project know different things than one another. That's inevitably always the case since everyone's strengths,

interests, and backgrounds are different.

Challenge 2: Abrasiveness

Sometimes shyness seems to evolve as an overcompensation for unsuccessful past interactions. Someone tried to speak up, to share their idea or input, just to add to someone else's point and yet it somehow wound up with hurt feelings and no difference in results. Entering into the discussion got people riled up, one too many times, so after one last time throwing hands into the air out of frustration, a developer decides to just stop trying. Maybe they feel that their input wasn't properly received, or even if it was it simply wasn't worth the trouble involved.

As one of my mentors in my undergraduate years pointed out to me, "Sometimes when people are fighting against your point, they're not really disagreeing with what you said. They're disagreeing with how you said it! If you made

the same point differently they might get behind it.”

He was absolutely right. Once I heard that idea, in addition to catching myself doing it, I began to notice it everywhere from others as well. It causes tension in meetings, collaborative classroom projects, even just everyday conversations between people. Well-meaning folks with no intention of being combative, indeed in total overall agreement about both goals and general means, often wind up in counterproductive, circular scuffles arising from an escalation of unintended hostility.

There are causes and patterns of behavior that lead to this problem. After 10 years of working on it, I’ve gotten better about this, but it still happens on occasion, and it’s still something that I have to actively keep ahead of.

It’s understandable how someone could run through this pattern only so many times before feeling like

their engaging with the group is the cause of the trouble. This is in turn followed by backing off, toning down their level of personal investment in the dialog, and (often bitterly) following orders from the action items that remain after others get done with the discussion.

IN PART 2: PRACTICAL STRATEGIES

In either case – shyness or abrasiveness – and in any role on a team, nobody gains from having one less voice of experience, skill, and genuine concern involved. Simply tuning out isn’t doing that person, their team, the game, or the players any real benefit. The issue isn’t the person or their ideas; the issue is just how the communication is performed, and just as with any other skill a person can improve how they communicate.

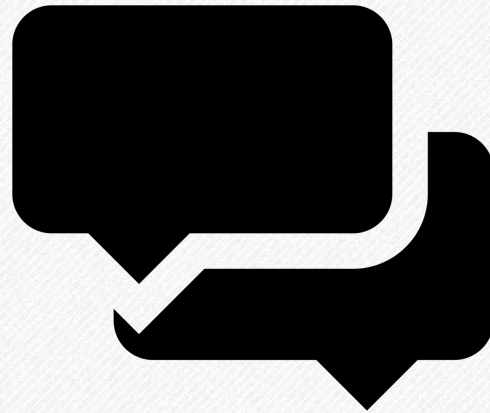
Failing to figure out a way to overcome these communications challenges can cause the team and developer much more trouble later, since not dealing with a few

small problems early on when they're still small can cause them to grow and erupt later beyond all proportion.

In the next section, we'll go over some practical strategies, more considerations, and outside resources that I've found helpful in making continual progress toward become a better communicator, and by extension, a happier and more productive videogame developer. My hope is that some of these will speak to some of the challenges that you or your peer developers are up against, too.

COMMUNICATION IS A GAME DEVELOPMENT SKILL, PART 2

In this next part we'll look through a mixed bag of tools that we can consider and utilize when trying to improve our communication with collaborators. We owe it to our teams (and friends) to polish our rough edges!



In the previous section, I identified a couple of common points of trouble that arise for videogame developers. Communication matters! If communication skills are in bad enough shape, no matter what other technical or artistic skills have developed, it'll be hard to get others to collaborate, and hard to get players and reviewers to hear about the work. On the other end of the spectrum, if you communicate masterfully enough – persuasively, clearly, fairly and

reasonably – then the scale and types of projects that it's possible for you to work on expands tremendously, because being a good communicator is essential to winding up as a meaningful part of (or even starting and leading) any team of other developers who are bringing their own respective skills, experience, and inspiration into the picture.

LISTENING AND TAKING TO HEART

You've heard this all your life. You'll no doubt hear it again. Hopefully every time

communication comes up this gets mentioned too, first or prominently.

Listening well, meaning not just hearing what they have to say or giving them an outlet but trying to work with them to get at underlying meaning or concerns and adapting accordingly, is way harder than it sounds, or at least more unnatural than we'd like to think. You can benefit from practicing better listening. I can say that without knowing anything about you, because everyone – presidents and interns, parents and kids, students and teachers – can always listen better.

There's a tendency, even though we rationally know it's out of touch with reality, to think of oneself as the protagonist, and others like NPCs. Part of listening is consciously working to get past that. The goal isn't to get others to adopt your ideas, but rather it's to figure out a way forward that gains from the multiple backgrounds and perspectives available, in a

positive way that people can feel good about being involved with.

DON'T CARE WHO WINS, EVERYONE WINS

There's no winner in a conversation.

This one also probably sounds obvious, but it's an important one that enough people run into that it isn't pointed out nearly enough. Development discussion doesn't need to be a debate. Even to the extent that creative tension will inevitably present certain situations in which incompatible ideas are vying for limited development attention on a schedule, debate isn't the right way to approach the matter.

In one model for how a dialog exchange proceeds, two people with different ideas enter, and at the end of the exchange, one person won, one person lost. I don't think that anyone consciously thinks about dialog this way, but rather it may emerge as a default from the kinds of exchanges we hear on television from political talking heads, movie

portrayals of exchanges to establish relative status between characters, or even just our instinctive fight or flight sense of turf.

Rather than thinking in terms of who the spectators or an impartial judge might declare a winner, consider which positions the two people involved would likely take in separate future arguments. Avoid spectators when possible, as it can often pollute an otherwise civil exchange with defensive, ego-protecting posturing.

If all of your prior references have led you to believe strongly about a particular direction, you only do that rhetorical position, as well as your team and project, a disservice by creating opponents of it. Whenever we come across as unlikeable, especially in matters like design, art, or business where a number of directions may be equally viable, then it doesn't matter what theoretical support an option has if people associate it

with a negative, hostile feeling or person.

Be friendly about it. Worry first about understanding the merits and considerations of their point, then about your own perspective being understood for consideration. Notice that neither of those is about "convincing" them, or showing them the "right" way. It's about trying to understand one another because without that the rest of discussion just amounts to barking and battling over imagined differences.

YOU MIGHT JUST BE WRONG

Speaking of understanding one another, don't ever be afraid to back down from a point after figuring out what's going on, and realizing that there's another approach that'll work just as well or better. There's a misplaced macho sense of identity attached to sticking to our guns over standing up for our ideas – especially when the ideas aren't necessarily thoroughly developed

and aren't exactly noble or golden anyhow.

A smart person is open to changing their mind when new information or considerations come to light. You're not playing on Red team competing against Blue team. You're all on the same team, trying to get the ball to go the same direction, and maybe your teammate has a good point about which direction the actual goal is in.

The other side of this is to give the other person a way out.

Presenting new information or concerns may make it easier for them to change their mind, even if that particular information or concern isn't actually why they change their mind, simply because it can feel more appropriate to respond to new information than to appear to have been uncertain in the first place. Acknowledging the advantages in the position they're holding doesn't make your position seem weaker by comparison, it makes them feel

listened to, acknowledged, and like there's a better chance you're considering not just your own initial thoughts but theirs too. When a point gets settled, whichever way things go, let the difference go instead of forming an impression of who's with or against you. Such feelings have a way of being self-fulfilling. In practice, reasonable people are for or against particular points that come up, not for or against people.

When an idea inevitably gets compromised or thrown out, being a skilled communicator means not getting bitter or caught up in that. Don't take it personally. It's in the best interests of the team, and therefore the team's members (yourself included), that not every idea raised makes it into implementation or remains in the final game.

BENEFIT OF THE DOUBT, ASSUME THE BEST

A straw man argument is when we disagree with or attempt to disprove a simplified opposition

position. In informal, heated arguments over differences in politics, religious, or cultural beliefs, these are frequently found in the form of disagreeing with the most extreme, irrational, or obviously troubled members of the group, rather than dealing with the more moderate, rational, and competent justifications of their most thoughtful adherents. This leads to deadlock, since both sides feel as though they're advancing an argument against the other, yet neither side feels as though their own points have been addressed.

When the goal is to make a more successful collaboration, rather than to just make ourselves temporarily feel good, the right thing to do is often the opposite of setting up a straw man argument. Assume that the other person is coming from a rational, informed, well-intentioned place with their position, and if that's not what you're seeing from what has been communicated so far, then seek to

further understand. Alternatively, even help them further develop their idea by looking for additional merit to identify in it beyond what they might have originally had in mind – maybe from where you're coming from it has possible benefits that they didn't realize mattered to you.

If the idea you may be holding is different than what someone else is proposing, welcome your idea really being put to the test by measuring it against as well put-together an alternative as the two of you can conceive. If it gets replaced by a better proposal that you arrived at through real discussion and consideration, or working together to identify a path that seems more likely to pan out well for both of you, all the better.

YOUR FRUSTRATION IS WITH YOURSELF

This is one of those little life lessons that I learned from my wrestling coach which has stayed with me well after I finished participating in athletic competitions. Most of the time

when people are upset or frustrated or disappointed, they're upset or frustrated or disappointed mostly with themselves, and directing that at somebody else through blame isn't ever going to diffuse it.

Even if this isn't 100% completely and totally true in every situation – sure, sometimes people can be very inconsiderate, selfish, or irresponsible and there may be good reason to be upset with them – I find that it's an incredibly useful way to frame thinking about our emotional state because it takes it from being something the outside world has control over and changes to focus to what we can do about it.

Disappointed with someone violating our trust? Our disappointment may be with our failing to recognize we should not have trusted them. Upset with someone for doing something wrong? We may be upset with ourselves for not making the directions or expectations more

clear. Frustrated with someone that doesn't understand something that you find obvious? Your frustration well may lie in your feeling of present inability to coherently and productively articulate to that person exactly what it is you think they're not understanding.

If your point isn't well understood or received but you believe it has value that isn't being rightly considered, rather than assuming the other person is incapable of understanding it, put the onus on yourself to make a clearer case for it. Maybe they don't follow your reference, or could better get what you're trying to say if you captured it in a simple visual like a diagram or flow chart. Maybe they understand what you're saying but don't see why you think it needs to be said, or they get what you mean but don't see the connection you have in mind for what changes you think it should lead to.

Clarify. Edit it down to summary highlights (people often have trouble absorbing details of an argument until they first already understand the high level). Explain it another way to triangulate. Provide a demonstration case or example. If there's a point you already made which you think was important to understanding it but that point didn't seem to stick, find a way to revisit it in a new way that leads with it instead of burying it among other phrases that were perhaps too disorganized at the time to properly set up or support it.

MISTAKING TENTATIVE FOR DEFINITIVE

Decisions can change. When they're in rough draft or first-pass, they're likely to – that's why we do them in rough form first! It's easier to fix and change things when they're just a plan, an idea, or a prototype, and the more they get built out into detail or stick around such that later decisions get made based on them, then the more

cemented those decisions tend to become.

There are two types of miscommunication that can come from this sort of misunderstanding: mistaking your own tentative ideas for being definitive, or mistaking someone else's tentative ideas for being definitive. During development, and as more people get involved, projects can change and evolve a bit to reflect what's working or what isn't, or to take better advantage of the strengths and interests of team members.

If there was an idea you pushed for earlier in a project and people seemed onboard with it then, it's possible that discoveries during development or compromises being made for time and resource constraints have caused it to appear in a modified or reduced form. It might even be cut entirely, if not explicitly then maybe just lost in the shuffle. Before raising a case for it, it's worth rethinking how the project may have

changed since the time the idea was initially formed, to determine whether it would need to be updated to still make sense for the direction the team and game has gone in.

Sometimes the value of ideas during development is to give us focus and direction, and whether the idea survives in its originally intended form is secondary to whether the team and players are happy with the software that results. It may turn out to be worth revisiting and bringing back up, possibly in a slightly updated form, as maybe last time was at a phase in development when it wasn't as applicable as it might be now. Or it may be worth letting go as having been useful at the time, but perhaps not as useful now, a stepping stone to other ideas and realizations the team has made in the time that has passed.

The other side to this is to make the same mistake in thinking about someone else's ideas: thinking they are definitive when

they are necessarily tentative. This happens perhaps because of how far off the idea relates to the future, and how much will be discovered or answered between now and then that is unknown at the time of the initial conception and discussion. If a project recruits people with the intention of supporting a dozen types of cars, but during development reality sinks in and only three different vehicles make sense in favor of putting that energy into other development necessity, those things happen. People get optimistic, people make planning mistakes, and people cannot predict the future – but it's important to not confuse those perfectly human imperfections with knowingly lying or failing to keep a promise. If early in a project someone is trying to spell out a vision for what the project may look like later, don't take that too literally or think of it as a contract, look at it as a direction they see things headed in.

Implementation realities have a way of requiring compromise along the way.

SOFTEN THAT CERTAINTY AWAY

A common source of fighting on teams is from a misplaced sense of certainty in an observation or statement which reflects value priorities that someone else on the team doesn't necessarily share, or especially when the confidently made statement steamrolls value priorities of someone else on team.

Acknowledge with some humility that you only have visibility on part of all that's going on, and that the best you can offer is a clarification of how things look for where you're coming from or the angle you have on things. Leave wiggle room for disagreement. Little opening phrases like "As best as I can tell..." or "It looks to me like..." or "I of course can't speak for everyone, but at least based on the games that I've played in this genre..." may just seem like filler, but in practice they can be the

difference between tying the team in a knot or opening up valuable discussion about different viewpoints.

CONSULTANT'S FRUSTRATION

School surrounds people with other people that think and work in similar patterns, with similar values, often of the same generation. That isn't typically how things work outside the classroom, whether collaborating on a hobby game project, joining a company, or doing basically anything else in the world besides taking Your Field 101. Often if your skills are in visual art, you have to work with people that don't know as much about visual art as you. If your skills are in design, you'll have to work with a lot of non-designers. If you have technical talents, you will be dealing with a lot of non-technical people.

That is why you are there. Because you know things they don't know. You can spot concerns that they can't spot. You understand what's necessary to

do things that they don't know how to do. If someone else on the team or company completely understood everything that you understand and in the same way, they wouldn't really need you to be involved. Your objective in this position is to help them understand, not to think poorly of them for knowing different things than you do. Help them see what you see. Teach a little bit.

I refer to this as the consultant's frustration because that's a case that draws particular emphasis to it: a company with no understanding of sales calls in a sales (or design, or IT, or whatever) consultant, because they have no understanding of that and that's why they made the call. A naive, inexperienced, unprepared consultant's reaction to these situations is one of horror and frustration – how on Earth are these people so unaware of the basic things that they need to know? The consultant is there to spot that gap and help them

bridge it, not to look down on them for it. Meanwhile, they're doing plenty of things right the specialist likely doesn't see or fully understand, because that's not the discipline or problem type that they're trained and experienced in being able to spot, assess, or repair.

When you see something that concerns you, share that with the team. That is part of how you add value. You may see things that others on the team do not.

VALUES ARE DIFFERENT PER ROLE

The other side to the above-mentioned point is appreciating that other factors and issues less visible to your own vantage point may have to be balanced against this point, or in some cases may even override it.

Frustration can arise from an exaggerated form of the consultant's frustration: a programmer may instinctively think of other roles on the team as second-rate programmers, or the designer may perceive everyone

else on the team as second-rate designers, etc. This is not a productive way to think, because it's not just that they are less-well suited to doing your position, but you're also less-well suited to doing theirs. A position goes beyond what skills someone brings to move a project forward; it also brings with them an identity and responsibility on the team to uphold certain aspects of the project, a trained eye to keep watch for certain kinds of issues. The programmer may not be worried about the color scheme, the artist may not be worried about how the code is organized, the designer may not care about either as long as the gameplay works.

That's one of the benefits of having multiple people filling specialized roles, even if it's people that are individually capable of wearing multiple hats or doing solo projects if they had to.

In the intersection of these concerns, compromises inevitably have to get made. The artist may be annoyed by a certain anomaly in how the graphics render, but the engineer may have a solid case for why that's the best the team's going to get out for the given style of the technology they have available. The musician or sound designer may feel that certain advanced scoring and dynamic adjustment methods could benefit the game's soundscape, but the gameplay and/or level designer may have complications they're close to about user experience, stage length, or input scheme that place some tricky limits on the applicability of those approaches.

One of the reasons why producers (on very small student, hobby, or indie teams this is often also either the lead programmer or lead designer) get a bad rap sometimes, as the "manager" that just doesn't get it, is because their particular accountability is to ensure that the game makes

forward progress until it's done and released in a timely manner. So the compromise justification that they often have to counter with is, "...but we have to keep this game on schedule" which is a short-term version of "...but this game has to get done." If someone isn't fighting that fight for the project, it doesn't get done.

Be glad that other people on a team, when you have the privilege of working with a good and well-balanced team, are looking out for where you have blind spots. Push yourself to be a better communicator so that you can help do the same for them.

TOO MUCH EMPHASIS ON ROLE

After that whole section on roles, I feel the need to clarify that especially for small team development (i.e. I can totally understand military-like hierarchy and clarity for 200+ person companies) roles shouldn't pigeonhole someone's ability to be involved in discussions and considerations.

While it's true that the person drawing the visual art is likely to have final say on how the art needs to be done (not only as a matter of aesthetic preference, but as a side effect of working within their own capabilities, strengths, and limitations), that does not mean that others from the team shouldn't be able to offer some feedback or input in terms of what style they feel better about working with, what best plays to their own strengths and limitations (ex. just because an artist can generate a certain visual style doesn't mean the programmer's going to be able to support it in real-time), and what they like just as fellow fans of games and media.

Does one team member know more about animation than others on the project? Then for goodness sake, of course that person needs to be involved in discussions affecting the implementation or scheduling of animation. But even if you're not an animator, if you've

accumulated a different set of media examples to draw upon, and have an idea for how that work may intersect with technical, design, or other complications, there's still often value in being a part of that discussion. Though of course you should still leave much of the decision with whoever it affects most, and whoever has the most related experience.

It's unhelpful to hide behind your role, thinking either "Well, I'm not the artist so that isn't my problem" or "Well, I'm the designer, so this isn't your problem to worry about."

The quality of the game affects everyone who got involved with making it. You make a point of surrounding yourself with capable people that are coming from different backgrounds and have different points of view to offer. Find ways to make the most of that.

A related distinction to these notes about roles is the concept of servant-leadership. Rather than a producer, director, or lead

designer feeling like the boss of other people who are supposed to do what they say, it can be healthy and constructive for them to approach the development as another collaborator on the team, just with particular responsibilities to look out for and different types of experience to draw from.

They're having to balance their own ideas with facilitating those of others to grow a shared vision. They're trying to keep the team happy and on track, and that's their version of the compiler or Photoshop.

HANDLING CRITIQUE PRODUCTIVELY

When critique comes up – whether of your game after it's done or of a small subpoint in a disagreement – separate yourself personally from the point discussed. When people give feedback on work you're doing, whether it's on your programming, art, audio, or otherwise, the feedback is about the work you're doing, it's not feedback about you (even if, and let's be fair here, we could all

honestly benefit from a little more feedback about ourselves as a work-in-progress, too!).

Feedback is almost always in the interest of making the work better, to point out perceived issues within a smaller setting before it's too late to fix the work in time for affecting more people, or before getting too far into the project to easily backtrack on it. Sometimes the feedback comes too late to fix them in this case, in which case rather than disagree with it, accept that's the case and keep it in mind to improve future efforts . After all, this isn't the last game or idea you're ever going to work on, right?

Defensiveness is often counterproductive, or at least a waste of limited time and energy.

SYSTEMS AND REGULAR CHANNELS

Forms and routine one-on-one check-in meetings can feel like a bureaucratic chore, but in proper balance and moderation, they can serve an important function. People need to have an outlet to

have their concerns and thoughts heard. People need to be in semi-regular contact with the people who they might need to raise their concerns with, before there is a concern to be raised, so that there's some history of trust and prior interaction to build upon and it doesn't seem like a weirdly hostile exception just to bring up something small.

In one of the game development groups I've been involved with recently, we were trying to narrow down possible directions for going forward from an early stage when little had been set into action yet. From just an open discussion, three of the dozen or so ideas on the whiteboard got boxed as seeming to be in the lead. When we paused to get a show of hands to see how many people were interested in each of the ideas on the board, we discovered that one of the boxed items had only a few supporters – those few just happened to be some of the more vocal people in the room. Even

introducing just a tiny bit of structure can be important in giving more of an outlet to the less outspoken people involved with a project, who have ideas and considerations that are likely just as good and, as mentioned earlier, probably weighing different sets of concerns and priorities.

PRACTICE, MAKE MISTAKES TO LEARN FROM
Seek out opportunities to get more practice communicating in all roles and at all scales, as part of a crowd or in front of a crowd, in both formal and informal settings, and with a few people or a lot of people.

Look for (or create) situations where you can comfortably exercise your communication abilities. Whatever form that may take for you.

Given a choice to work alone or work with a group, welcome the opportunity to deal with the challenges of working with a group. Attend a meetup. Find some clubs to participate in. When your team needs to present an

update, volunteer to be the one presenting that update.

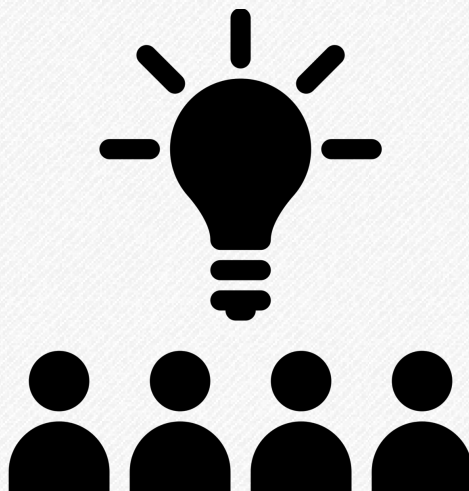
Communication is a game development skill. As with any other game development skill, you'll find the biggest gains in ability through continued and consistent practice.

RECOMMENDED READING

- *How to Win Friends and Influence People* by Dale Carnegie (1936)
- *The 7 Habits of Highly Effective People: Powerful Lessons in Personal Change* by Stephen R. Covey (1989)
- *To Sell Is Human: The Surprising Truth About Moving Others* by Daniel Pink (2013)

ESTABLISHING A VIDEOGAME DEVELOPMENT CLUB

Having a community of people with skills and goals can significantly lower the barriers to larger, higher quality games. I established two game clubs, and I really want you to establish one, too. Here's how I did it.



Carnegie Mellon's Game Creation Society

While in school, I helped get a game development club off the ground, structuring its processes, setting the standards, and ensuring projects began in a way for their teams to finish. That club is the Game Creation Society at Carnegie Mellon, and you can find the videogames from it at GameCreation.org.

I have since followed this same general pattern and structure years later as a graduate student

at Georgia Tech to establish VGDev.

Although that game development club was started at a university setting, many lessons learned from it are equally applicable to kicking off and growing a community of game developers, whether locally or online, whether as a high school club or among people long finished with school.

This is What Worked for Us

Every year, many videogame development clubs start. Many of

them initiate a handful of large projects, struggle with teams falling apart as member retention suffers, and after a couple years of sliding schedules, either close down or transform into a game playing or culture club instead of game development club.

Of course, what worked for us certainly isn't the only way to do things. There are surely other organizations out there that have succeeded, doing things very differently and following different rules of thumb. All I can say for sure is that my rules of thumb worked well for us during the first few years. I'm hopeful that others with drastically different experiences will help share learnings from their organizations, as well.

Lessons Learned

IF YOU DON'T BUILD IT, THEY WON'T COME

Although I developed the process for the group, and led the Game Creation Society for two years, I did not start it. Curt Bererton, then a Robotics PhD student and now

the founder and CEO behind ZipZapPlay, Curt founded the organization, writing its charter, posting flyers, building members, and running weekly meetings before he found any of us on campus that helped the group take off. He didn't kick off the organization with all of the people and resources that he needed – he kicked it off partly as a way to find them.

KEEPING PROJECT SCOPE WITHIN ONE SEMESTER

Each semester, everyone's time commitments changed. The longer a project took, the greater amount of uncertainty was involved between starting and finishing. For these reasons, we attempted to schedule every project to fit within the project it was started in. Doing a more complicated project was only possible by working faster, devoting more hours to the project, and being clever about avoiding unnecessary or wasted effort. For the especially ambitious projects, we occasionally would

let experienced developers plan for a full release at the end of one semester, then a second release with additional features and content the next semester.

EMPHASIZE COMPLETING ALL GAMES

Finishing every project started is terribly important. This is true even if it means encouraging less ambitious project scopes, shorter timelines for greater predictability, and turning down some enthusiastic “I want to make the next big MMO” types. Nothing is worse for new member recruitment than having a spotty track record of incomplete projects, and nothing is worse for member retention than having art, audio, and levels made but thrown away. New members are attracted by the prospect of working with a reliable team, using a known process, in high confidence that their work will result in a finished game.

VET PROJECT LEADERS

Each project leader had to fill out a simple one-sheet proposal,

outlining what the name of the game was, how long it was anticipated to take (again: maximum one semester), what it was about in a nutshell, and how many people of each position the project would be expected to need. We then met one-on-one for an hour or so, chatting about the project’s goals and schedule. This wasn’t a very strict or demanding filter, but it was an added layer of protection to help minimize the chances of otherwise motivated members being brought onto dead-end projects started by well-intentioned but unprepared project leaders.

IT’S OKAY TO LOSE PEOPLE THAT WON’T WORK

Making videogames involves a lot of time, energy, work, and additional learning. Each year, we brought in as many members as we possibly could during the Spring Activities Fair – with as many as 100-150 people showing up at our following meeting – by focusing on the importance of having all kinds of people (artists,

writers, programmers, modelers, musicians, testers...) involved in making videogames. In the first meeting or two, we explained the process, and some people would leave when they understood that this would involve more time each week than just attending the 1-2 hour meetings, and wasn't the same as simply playing videogames. This was a good thing, not a problem.

LEADERS PITCH, MEMBERS VOLUNTEER

The project pitch was just a 15 minute powerpoint (showing mock up screenshot, spelling out the project concept or similar games, etc.) and speech about the project leader's initial vision for the project. The last part of the project's pitch was the leader's name, e-mail address, and a list of the minimal roles they need for the project to be made: two 2D artists, one sound person, etc. Anyone interested in those roles would then talk to the project lead after the meeting, or contact them by email. If the project leader didn't

collect enough members within 2 weeks of being pitched, or didn't adjust its plans to reflect the members that did join, then it wouldn't become an active project. When people aren't being paid, and aren't receiving credit, its important that they're able to work on something that appeals to them.

SCHOOL CREDIT THREATENS AGILITY

One of the advantages of being a non-class and non-workplace organization is that people who are disinterested can just leave, instead of being locked in by their need for the salary, school credit, etc. At first we began to investigate the opportunity for active students to get course credit or some other type of academic recognition for their work in the organization, but after a few semesters we were thankful for the agility that came with members self-selecting based purely on their interest in making videogames.

REACH OUT TO GUEST SPEAKERS

There are a collection of people in the organization that are eager to hear about videogame industry topics, ranging from recruitment talks to more experienced developers sharing their insights. Meanwhile, companies have people who devote their full attention to finding an audience to recruit from, and experienced developers are often happy to have a chance to share their stories. Establish email contact with anyone and everyone that you think might make the trip – more people will say no than yes, but even one yes a year can translate to happier members, new members (guest speakers are prime events to invite non-members to drop in on), and improved networking opportunities for all. It can even lead to internships and jobs for organization members, which translates into lasting alumni industry connections.

PROVIDE OPTIONAL TEAM RESOURCES

We gave every active project team a folder on an FTP server with a custom login and password, for members on the team to share files, as well as a section on a message board to keep a history of each project's discussion. These were provided as free secondary user accounts on a low-cost StartLogic.com account, using ezBoard then phpBB for the bulletin board system. These minimal resources helped facilitate working around asynchronous schedules for student teams.

Between Google Docs, Dropbox, Assembla, and GitHub, and other services already out there, it may not be as useful anymore for the club to provide things like dedicated FTP space.]

GIVE CONTENT CREATORS LATITUDE

People want to show off, and get practice in, whatever style or subject they know how to do best. Projects that can accommodate this will come out better, yielding

not just better games but happier members.

WEEKLY GOALS FOR EVERYONE

Busy work isn't a good thing, but people get involved in hobby projects out of an interest to contribute, gain experience, and have something to show off proudly when it's done. If someone on the project team has a three week period with nothing to do, they may fade away from feeling like there's no use for them on the team.

ALLOW PROVEN MEMBERS TO TAKE ON MORE

This partly addresses the above problem – while one project's need for an artist or audio contributor slows down, another project's needs may swell. This also creates a greater variety of experience in the same amount of time, and helps project completion.

ENCOURAGE HISTORICAL PROGRESSION

Although everyone's temptation is to jump right into the deep end, making games that resemble the ones they've been playing lately, that frequently leads to people

biting off more than they can chew. The game either doesn't get finished, or it doesn't get finished well.

Instead, I've encouraged club members to think historically for reference points, as a way to tackle less complex (but fully complete) games before moving on to more involved ones. The best way that I've found to get this across is to frame it in terms of decades or half-decades.

Encourage developers that have never made a game before to start or join teams making games of 1970s complexity (granted, they may look and sound more modern – this is only about how involved the code and design become, and the game's overall scale or scope). After they've led or been involved with a team of 1970s, move on up to 1980s or early-1980s complexity, then up toward 1990s, and so on. This gives clear reference frames for what's appropriate and mountains of enjoyable examples to draw from.

More projects get finished. More projects get finished well. Most importantly, more developers get the foundational skills and production experience that they need to later see through some of their more ambitious ideas together.

DON'T BECOME A GAME PLAYING CLUB

We had members and non-members alike ask our organization to host tournaments, game nights, game release parties, retro game nights, game movie nights, and every such variation. There's nothing wrong with those types of events, but as mentioned at the start of this entry, one of the common causes for organizations of this sort losing focus and getting nothing done is on account of becoming a game playing and culture club instead of a game development club.

The videogames developed will be worse, and more likely to wind up unfinished, if a bunch of people are hanging around not for their dedication to game making but for

their dedication to game playing. Let those students find such outlets elsewhere. There's no shortage of opportunities to play videogames with other people in college.

If any game playing goes on under the organization's name, it ought to be playing games completed by the club, either to help new members see the types of projects that can be done in the timeframe given, or to help recruit outsiders at a public demo by showing that the club is serious about creating finished, playable games.

ACTIVELY RECRUIT A VARIETY OF PEOPLE

A room full of people that are experts in the same area will all overlook the same problems and weaknesses. Every type of expert can help fill in for different oversights by others.

Even if it's a room full of computer science people that know how to program (this was our core group with the Game Creation Society), it helps to find programmers with a variety of side interests in music

composition, sprite animation, interface design, etc.

Active recruitment is also an important aspect of this. Flyers are important, but evangelizing for the cause goes a long way, as does being outspoken at fairs and other recruitment opportunities.

BE TECHNOLOGY AND TOOL AGNOSTIC

Some 3D artists prefer Maya to 3D Studio, or vice versa, while some others are most productive in Blender (or they may not have \$3,500 laying around to spend on their hobby). Some programmers swear by C#, some prefer ActionScript 3, while others are content to script in Unity or Game Maker.

Whatever tools and technology platforms that people know best, welcome them to use those platforms to their fullest extent. At the end of the day happy members and well-developed content will go further. Giving people freedom to teach themselves new things is awesome – but forcing them to

teach themselves new things is what their class time is for.

MINIMIZE CONTENT BOTTLENECKS

Role-playing games and adventure games are scary. No matter how simple they look, they require (at a minimum) dozens of pieces of enemy art, level layouts for dozens of cities, outdoor areas, and dungeons, balanced stats, prices, and art for a wide variety of items, several emotionally intense songs, and countless pages of strangely paginated written dialog for dozens of characters. It probably needs at least a minimal in-game scripting engine, too.

After I had been making videogames for seven years, I was able to lead a small team to make a really weak, limited RPG. That isn't offered as encouragement, but as a warning.

Side-scrolling beat-em up games and platforming games also involve a huge amount of animated art. Variety in bad guys, bosses, and backgrounds are

what compel people to keep playing them.

Some types of videogames scale much better, and don't require mountains of different art – especially puzzle or action games that take place in non-scrolling/one-screen levels. When beginning teams present their first projects, it's best to favor games that will be able to efficiently reuse content in different combinations, and get away with minimal content. Not only does it improve the chances of that particular project succeeding, but it prevents any one overly ambitious, epic project from gobbling up all of the club's content creators into a black hole that (likely) won't turn into a finished game.

PROJECT LEADERS MUST BE ABLE TO FINISH ALONE

People will leave the organization. People will leave teams. No matter what people intend, and no matter how nicely everyone is treated, things will change, and priorities will shift. To the outside world, it's

very important that the club maintains a reputation for finishing its games. Internally, this means it's very important to the club that project leaders ensure that the game they start gets finished. This requires that project leaders have at least minimal fluency in digital art, programming, and making audio for use in the game, such that if everyone else left the team, it could be finished, even though it may not look or sound as good as was intended. This accountability further motivates project leaders to keep their project members content, and to seek replacements any time project members leave the team, since work they can't get someone else to do is work that becomes theirs.

THE DEMO IS THE GAME

As a scheme for scope reduction, one thing we did to help reel in project scope during pre-pitch meetings was to ask the project leader what is the minimal amount of functionality and content they

would need to create a demo for the game.

Since it's a portfolio and experience piece for the team's project members, this demo size is about as much content as most outsiders will digest. Making a larger version of the game would have diminishing returns on learning, in addition to increasing the odds of the game never being completed. And a videogame that isn't completed isn't a videogame to the outside world.

In consideration of this, we adjusted plans, content, team size, and story for the "demo" sized version to be the finished game, before the project was ever pitched or started.

This has the unintended side effect of increasing believability and buy-in from members to the project pitch. When someone tries to entice members to join their project that is planned to have 20 worlds, 30 weapons, and 16 main

characters, no one takes a pitch like that seriously.

FOCUS ON VIDEOGAMES, NOT GAMES

A club that tries to be for everybody is a club that works for no one. A board game, dice game, tabletop game, D&D game, casino game, sports game, game show, game theory, or Conway's game of life club is not a videogame development club. Even if those are things that may appeal to some categories of videogame designer, they're less likely to appeal to 3D artists, software engineers, dialog writers, playtesters, animators, music composers, sound effects specialists, and the whole host of other skills that go into hobby and professional team videogame development.

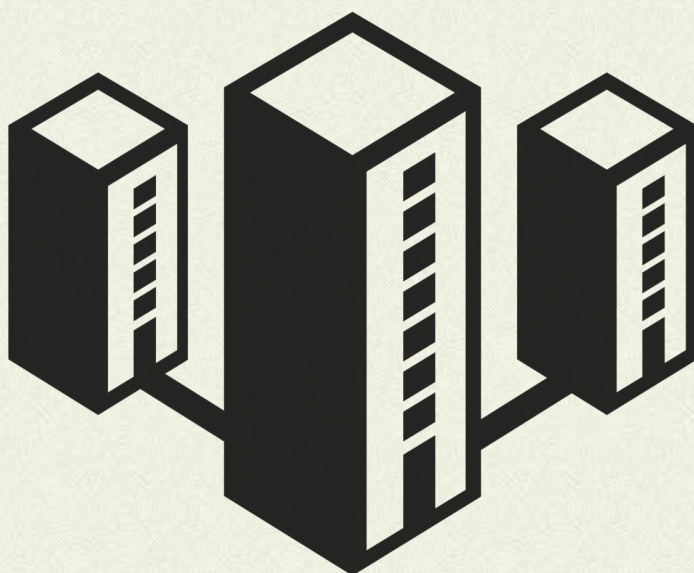
Welcome members that have an interest in those things, just as much as members are welcome that have interests in programming operating systems, animating feature-length films, and writing novels. But keeping it clear that

the organization is not there for pitching, producing, and networking games other than videogames, just as much as it isn't for film making or novel writing, helps keep the core strong.

BE READY TO ADAPT

Depending on who the organization's leaders are, what the project leaders want to make, the rules of thumb will need to be different. How established the club is will also lead to huge changes in organizational needs – a young club is battling aggressively for growth, credibility, and clarity, whereas a more established club does well to focus on retention, visibility, and stability.

8



Many people making their own videogames consider going the professional route. Here I'll share some stories and realizations from my professional years about considerations and tradeoffs to navigate. Every person's path is different.

INDUSTRY

A FRANK LOOK AT MAKING VIDEOGAMES PROFESSIONALLY

Many people that want to make videogames. This desire often begins from an interest in making their own game ideas. Over time, that interest tends to quietly morph into an interest in getting a job making games.



Why Even Discuss This

As the speaker Bill Gove used to say, “You’ve got to tell people the truth. That’s the only way to give people a fighting chance.”

My aim here is not to tell anyone what to do or not to do, but only to provide some additional information to consider about the way things often fit together – acknowledging too that partial exceptions, while rare, surely exist.

I’ve seen some smart people spend years getting into the industry, to then soon wind up frustrated and disillusioned, changing career paths after spending less time doing it than they did moving toward it. I think this often happens at least in part because of the misunderstanding that I’m going to examine here.

The Main Point

People that do an otherwise fun activity at an expert, competitive level have to engage with it in a

very different way than others who are not counting on it as their primary income. This is true for athletes, singers, hot dog eating champions, painters, eSports players, and dancers.

It's true for making videogames, too.

Some Implications

1. COMMERCIAL GAME DEVELOPMENT INVOLVES A BUNCH OF WORK THAT ISN'T DIRECTLY MAKING THE GAME.

When we think about what people or skills are needed to make a game, at minimum, we often split up our thinking into roles like code, design, art, and audio. Someone may be wearing multiple of those hats, and there may be need for a separate producer if the team has more than a few people total. However as long as those areas are covered a small game can get made.

For the game to exist properly as a commercial product, there are other business, legal, and marketing responsibilities that enter the picture: incorporation,

business taxes, pursuing funding (if applicable), contracts, engagement with fans and press, and so on.

With a larger company, those other positions outside of development tend to be filled by separate specialists. At a smaller company, those obligations often tie up a surprisingly large fraction of the time, attention, and energy of someone that's also trying to juggle one or more of those developer roles.

2. MANY PEOPLE THAT WORK AT LARGE GAME STUDIOS ARE UNABLE TO (OR NEVER HAVE) MADE A FULL GAME OF THEIR OWN.

I found this mind boggling when I first realized it.

A large professional team, unlike a small non-commercial or typical indie team, is largely filled with specialists. Those specialists may be world-class experts at what they do, but what they do is often a comparatively small piece of everything that comes together from dozens of other specialists: model rigging, scene lighting,

network backend, dialog writing, implementing someone else's design spec, texture artist, outsourcing coordination, management layers, and so on.

This is not at all to discredit, disrespect, nor devalue what professional videogame developers do. Altogether the coordination of specialists is essential to making some truly amazing end products possible, which many players enjoy. Clearly, a simple capitalist argument can be made that they are demonstrably among the very best people at what they do. They are all, as a part of the total team, professional videogame makers.

Yet surprisingly few of them have ever worked on something that was truly their own personal game, their own original idea, or their own direction. They may have worked on dozens of commercially successful games that you've heard of, however on each game their individual input or exact contributions to the overall

project may be difficult to pinpoint. It couldn't have happened without them – or someone with a similar skillset filling in for them – but their work seamlessly blends into the whole.

That leads naturally to our next point.

3. LARGE COMMERCIAL GAMES OFTEN DIVIDE WORK AMONG MULTIPLE SPECIALISTS, YIELDING LESS CLEAR ARTISTIC OR AUTHORIAL OWNERSHIP.

On a small team, one person might think up how an enemy should look, then do the concept art for it, 3D model it, texture it, rig it, and animate it – maybe even code its behavior in-game, too. That's a lot of creative freedom, and in some ways amounts to being one's own boss since work earlier in the process affects what the work is like further ahead. In small enough projects the same person may even be doing that for every enemy, item, and environment piece in the game.

By contrast, on a large team, the person or people best at each step of the art process (say,

texturing), and/or for a particular kind of art asset (environmental, character, UI, etc.), may be working only on that slice of the process, forming a digital assembly line of sorts. The end result may be more impressive, as though the output of someone incredibly talented, well trained, and inhumanly consistent in every step of the process. It requires more sophisticated collaboration and scheduling. However this also shifts the nature of personal ownership from “I created these whole things within the game” to “I helped fill in aspects of many different things in the game.”

When students come fresh out of schools that have game development classes and join large studios they are often going from a background in which they enjoyed substantial ownership over a large fraction (often all) of one or more smallish personal projects, into a new position where they’re being paid to concern themselves only with one

very specific aspect of a far grander and longer term project. That transition can be a comfortable, healthy, desirable one for some people that prefer to specialize, especially if they see it coming and understand the implications. It can be a very difficult and unexpected change for others.

4. CREATIVITY WITHIN LARGE COMMERCIAL PROJECTS REQUIRES GOOD PROCESS, NOT FOLLOWING WHIMS.

With so many people’s work interconnected, the misuse or misdirection of time by one person can have a ripple effect on dozens of others on the team. This can make it terribly expensive to simply mess around, or to pursue our pet ideas within the project.

Running a business often involves a lot of processes, quality control, and office politics. Together these serve to filter, refine, and sometimes dilute individual contributions. This means conducting market research, having rigorous approval and testing processes, ruthless editing

for quality, and on large teams, many team members having relatively little leeway or input outside of their specific area(s) of expertise.

While potentially frustrating from the inside, to the outside world this can ultimately yield a product more likely to fare better in the marketplace. A game that's a mishmash with dozens of people's divergent interests and personal ideas about matters beyond their main skillset is likely to lack focus, coherence, and the level of fidelity that comes from specialists sticking to their specialties.

When people occasionally write war stories about their time in industry, with regard to burning out or finding it exhausting, beyond the total number of weekly hours worked I think it's often also because of how this process tends to grind up, pulverize, and over time wear down creative impulses. An environment must be fostered in which only the fittest ideas survive to reach customers.

When the outcome affects the salaried jobs of many people, having that filter process built in makes a lot better sense than just making whatever exploratory stuff comes to mind.

5. BUSINESS PRESSURES HAVE TO AFFECT COMMERCIAL GAME DESIGN.

When making games professionally there's potential for frustration and conflict all along the way between what you personally feel like creating and what you (or your managers, or backers) think is most likely to be the profitable decision.

If entertainment business worked on a linear scale, and was more predictable, there might be more room for someone to simply opt to make some conscious trade offs between making a little less money in exchange for some personal creative calls. However it's not a simple slider. If dealing only with profits from internally designed projects, as opposed to buffering it with outside contract work, the decision doesn't fall

between “do it our desired way and maybe earn a little less” but instead it comes down to “do what we think will be profitable, to hopefully stay in business longer” versus “ignore what we think will be profitable, and risk going out of business.”

The videogames business, like many professional entertainment domains, tends to be hit-driven, and public taste is a rapidly moving, unpredictable target. A very small percentage of participants do exceptionally well, meanwhile a much larger percentage scrape by, or even work at a loss, just for a shot at joining that small percentage at the top.

In that kind of environment, a creative call outweighing business pressures is so rare it's deemed newsworthy. In practice, even then it's at least partly a calculated, justifiable gamble to stand out, or to generate discussion, like the “No Russian” mission in *Modern Warfare 2*.

This isn't just about small detailed decisions, either. Some people get into making games because they are drawn to a certain gameplay genre to only later realize that by the time they're making games professionally that entire genre may be over saturated, widely deemed outdated, dominated by entrenched companies that had a massive head start, or otherwise has become a very unwise domain in which to focus a business.

Or, as a matter affecting many people currently in industry: some developers were drawn to the types of games designed around an older payment model. Then when the ways that people primarily pay for their games subsequently changes which types of games do well in the marketplace, many find it hard and unnatural to rethink the design ideals that they grew up with in a different way which is, for example, compatible with free-to-play, ads, in-app purchases, social

network integration, crowd-funding strategy, etc.

I'll never forget something that I learned from the first professional Lead Game Designer that I met: when he gave talks to students about making videogames professionally, he challenged them to think about how they'd go about the game design for Jar-Jar Binks Racing. He picked that case to illustrate that even as the Lead Game Designer, at a large company there are often business people higher in the chain deciding what can get made at a profit, and your task is to make the best version of that you can.

It's Still a Perfectly Fine Job

I think it's important to point out that most of the above is probably true about essentially any office career.

My point here is to remind people that working on videogames as a career, especially at medium or large companies, is also an office job (or at a solo and small team

scale, faces the same profit pressures and visibility challenges as any garage band or beginning novelist, but that's mostly a topic for another day). It has to be, in order to consistently cover the overhead of the workspace and people's salaries.

It certainly still takes talent and hard work, and often involves working with excellent coworkers. It challenges people's skills, creativity, and interpersonal maturity. It's a very hard thing to do at that scale. It's just of a fundamentally different nature at that scale, perhaps like how preparing for and playing soccer in the World Cup likely involves a lifestyle and stress level quite unlike playing a pickup game at the park with friends or neighbors.

I am not trying to talk anyone out of doing it. There are people who deeply love the work in those environments. For someone that desires deep specialization, instead of being involved with all

parts and details, it may be just what they want to do.

It Doesn't Have to Be a Job

You're not a failure or wannabe if (or while) you do it non-commercially.

There's sometimes a presumption with videogames that if you aren't doing it full-time to make your living, you're somehow illegitimate. I've heard the term "hobbyist" used derisively in developer communities, as in, saying someone is "just a hobbyist" to devalue what they make and do. That discussion wasn't about me or my work, but my repulsion over hearing the word used that way was actually a big part of what inspired me to name this blog and my Twitter handle HobbyGameDev.

If someone bowls on weekends, we don't demean the activity by saying she is merely "trying" to bowl, or is "just a hobbyist bowler." If someone likes to run

and does so often, we don't think he's failing at it if it isn't his career.

If you bowl, you're a bowler. If you run, you're a runner. If you develop videogames, you're a videogame developer.

As long as you're able to find ways to create games in your spare time, using resources that are free or priced for everyday people, you can make anything you want. Want to make a game with a dragon that flies through space? Want to make a fast paced puzzle game with mechanics you've never seen before? Want to invent entirely new vehicles and let people test drive them?

You don't need an industry job to do that. In fact, getting an industry job may not even lead to the opportunity to see your vision through, much as the hundreds of crew people that work on Steven Spielberg's films are unlikely to ever become the next Steven Spielberg – doing what they're

getting paid to do is generally not preparing them to replace him.

There are, granted, some people like him who are in positions in the entertainment industry where they can lead armies of others to realize their creative visions. Those positions are comparatively few (total, in the entire known universe) and many, many times harder to wind up in or hold than the already fiercely competitive entry level positions. That's true though only if you are doing it as a career. In contrast, if you're open to working on it as a hobby, you can start working on bringing your own creative vision to life literally right now, with the very same time and computer that you're instead using to read this article on the internet.

The result might not look or sound like a Next Gen title, pushing the limits of the latest tools and hardware. It probably will not get shown at E3. You may have to get clever about finding ways to pull it off with your capabilities and collaborators. But most

importantly: **you do not need anyone's permission to start making it, or to begin looking up and practicing the missing skills you'll need to pick up in order to do so.**

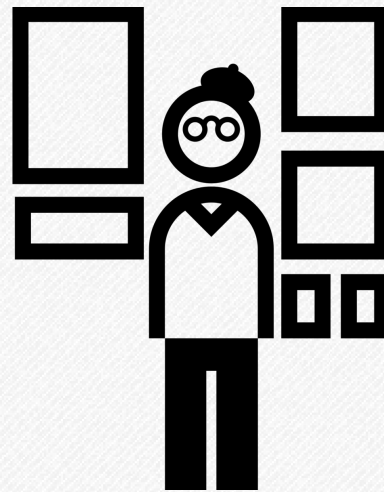
Maybe some people will fall in love with what you're making. Maybe people will urge you to take it to another level, finding ways to make a living from working on improvements to it. Or maybe they won't, in which case here's the worst-case scenario: you'll just be left with the game that you personally wanted to create. Along the way you also got practice at development skills that you can then apply to make your next projects better.

That's not a bad outcome.

In this section, I mostly distinguish stable careers within medium-large companies, in contrast to non-commercial small teams. Small teams making commercial projects operate in every form along the gradient between).

INDIE GAME DEVELOPMENT AS A CAREER

I stayed afloat as an indie game developer for several years, completing numerous projects and picking up a variety of experiences along the way. It was a very different experience than my brief time in AAA games.



Matt D., as part of an assignment, had to send questions to someone in a career he finds interesting. He chose independent game developer as the career, and I'm the someone he reached out to. Here were my replies, as of 2012.

1. WHAT IS THE PRIMARY MISSION OF THIS ORGANIZATION OR PROFESSION?

One of our key advantages as indie developers is our agility – we can change our reasons or approaches on a moment's notice without needing to retrain a hundred person workforce or get buy in from outside third parties, so the primary mission not only changes from one indie to another,

but also within the same indie from one project to another.

I don't see myself as having one primary mission, so much as having an evolving skill set that I'm continually looking for new challenges I can adapt it against: research projects, art projects, commercial projects, hobby projects, etc. each of which comes with its own mission particulars.

Sometimes an independent game developer's mission is to make

money, sometimes to change how people see or think about videogames, sometimes to express ourselves artistically, and sometimes to simply entertain folks by making something fun. I could give a very specific answer to my primary mission on any given project that I've worked on, but on the whole across all projects I don't believe there's a truly common thread.

2. WHAT ARE THE RESPONSIBILITIES OF YOUR DEPARTMENT?

When I have done technical game design for larger companies (Electronic Arts, ZipZapPlay, Will Wright's Stupid Fun Club) within a department, my responsibilities were much more narrowly defined than they are in my role as an independent game developer. In those cases I was responsible for digital prototyping, as well as proposing and documenting gameplay systems, laying out level content, and authoring gameplay elements.

Likewise, to a lesser degree, when I worked with publishers (ngMoco for Topple, Sonic Boom for Alice in Bomberland and iZombie: Death March), certain responsibilities were taken on by or at least shared with the publisher, such as creating marketing materials, managing QA testing, completing localization for other languages, etc. However unlike when I worked within a department at a large company, when my role was that of an outside contractor, I had the added responsibilities of scheduling work, vetting and signing subcontractors to collaborate with, handling my share of the budget, and managing relations with the publisher as an outside party.

However as an independent developer, I don't have a department to work within, I'm just a person making things. In this context, my response to question 3 is really the answer that fits here, too, so let's move on to that one:

3. WHAT ARE YOUR RESPONSIBILITIES?

I'm responsible for:

- Director work: deciding platform, business model, genre, etc.
- Production work: scheduling how long I have, what has to happen by when
- Designing: figuring out the details of what I'm trying to do
- Engineering: making the implementation work
- Content authoring: creating images, sounds, and levels
- Managing: when I have a team, for example I find someone better at animation or music composition to help create those assets. It's up to me to ensure they're productive, on schedule, and satisfied with their involvement
- Promotion: recording and editing promotional videos, making icons, writing descriptions, and otherwise trying to call awareness to the game
- Budgeting (when applicable; I make many projects for \$0)

- Hiring contractors (when applicable; no budget = no hiring)

4. HOW DOES YOUR JOB RELATE TO THE OVERALL ORGANIZATION?

Since, in a way, in the case of being an independent videogame developer my job is the overall organization, to provide a more interesting response to this question I'll instead address how my job relates to the overall ecosystem/industry of games.

Publishers look to people like me to execute on an idea, within a certain budget (I have lower overhead than a full game studio) and within a certain timeframe (I work extremely quickly), or as someone to place a bet on (they help fund a project, giving me more budget and feedback to work with than I'd have on my own, in exchange for a share of revenue and credit).

Players (by no means all, but certainly some!) value people like me as a source of game types that they're not likely to see come out of a big studio. I'm able to

experiment constantly and severely, without needing to explain much to anyone until it's done, at which point the end result is available for all to see, play, and sometimes borrow ideas from to build upon in other projects.

Indies also look to one another for advice, feedback, and success stories, with countless varied definitions of what success means (making money, getting noticed, creating something new that gets cloned, reaching a different audience, getting a game out on a particular device or distribution channel, even just finishing and releasing a complete game project).

I've accepted as one of my personal objectives getting more people into videogame making, which I do with my business, the student game development clubs I established (at Carnegie Mellon and Georgia Tech), and fielding developer questions by email as much as time allows.

5. WHAT OTHER PEOPLE DO YOU WORK MOST CLOSELY WITH AND WHAT DO THEY DO?

Although some of my projects are created alone, when I do work with others, I tend to work with completely different collaborators on nearly every project. The most common roles I've hired others for have been artwork (animated sprites, character illustrations, background paintings), audio (professional-quality sound effects, and in a few cases original soundtracks), and on non-commercial projects I tend to open up design roles to peers to help them get experience.

When I've worked with publishers I've typically had one to three main contacts at the company that I've spoken with regularly about progress, questions, feedback, and so on. See response 2 for a more complete explanation of what they do.

Lastly, in a very real but more indirect way, I work the players that try my games. That happens by keeping metrics of how many

people each project reaches, googling to find out what players are saying about it, reading all feedback I receive after it has been completed, and so on. All they have to do is try the game, and if it leads them to say something about it to either me or the communities in which they play, then they've done their part. This in turn helps give me a more complete picture of what sort of an effect the game is having (maybe it's played by many people, maybe it's only reaching a smaller crowd but they seem to really enjoy it, or maybe it flat out missed the mark and no one seems to like it but me!), and also helps inform me of how different player communities on the web respond to different types of videogames.

6. ARE COMPUTERS USED ON THE JOB? IF YES, IN WHAT CAPACITY?

Yes, computers are absolutely used on the job. They're used for programming, level design, image creation, audio editing, music composition, video editing,

communication between team members, documentation, financial/scheduling work, and pretty much every aspect of the process. In the end, the videogames are of course also played on computers.

7. WHAT TYPE OF EDUCATION OR TRAINING DOES THIS JOB NEED?

Because there's no hiring manager to impress when working independently, in theory all that matters is that someone has or can partner up to account for all the technical and creative abilities needed to pull a complete videogame together. However in practice, a good computer science degree is often helpful for videogame programmers to learn many useful ways to think about programming and solve novel problems, whereas proper training in creating digital art, editing audio, or other content production areas can be beneficial to preparing a videogame artist or audio person to create top-tier work.

Those traditional paths also offer the benefits of a potentially stronger network of peers to collaborate with while learning, additional opportunities to learn or first gain outside work experience through the institution's reputation and visibility, and of course a solid backup plan in case being an independent videogame developer doesn't work out full-time.

Although a very small circle of independent videogame developers have made a tremendous amount of money, for many it doesn't pan out financially, or works decently enough for a while but doesn't turn out to be sustainable as a primary source of income for a family.

8. WHAT TYPE OF EDUCATION OR TRAINING HAVE YOU HAD?

I completed an undergraduate Computer Science degree from Carnegie Mellon in 2007, with a minor in Business Administration.

I completed a Masters in Digital Media at Georgia Tech in Spring 2012, though that happened after

my professional independent videogame development.

My various work experience at/for other companies in many ways factors in as additional education and training for the type of work that I do.

Otherwise I've been making videogames in my spare time since the late 1990s, completing an average of four games every year since.

9. HOW DID YOU DECIDE ON THIS TYPE OF WORK?

I was comfortable with programming when I started college, so I assumed I would do that for a living, and went into Computer Science. On the side I helped launch a student videogame development club, and some of the industry guest speakers that we brought in accepted resumes. I got a call from one, which led to an internship, and I've been doing videogame work commercially in varying capacities since.

My transition from corporate to independent game developer happened in a number of steps, beginning with corporate studio work out of university, followed by getting involved with a videogame startup, over to doing videogame development on contract on the side, then to videogame development for a publisher, and lastly doing videogame development completely independently. Each transition occurred due to feeling like I was ready to entrust myself with taking on a larger chunk of the responsibility than I was able to do in my current (soon previous) position.

10. WHAT DO YOU SEE AS THE DEMAND FOR JOBS LIKE YOURS IN THE FUTURE?

It's really anyone's guess. The economics driving videogame development have changed dramatically every 5-10 years since the industry started, and it seems very likely to continue changing, possibly even faster. This is doubly true for independent videogame developers, which

existed semi-briefly for PC games in the early 1980s before studios largely crowded that space, then mostly disappeared for awhile as console manufacturers and PC retail also favored studios over individuals, but has re-emerged in the past 5-10 years or so for web games, smartphone games, digital distribution like Steam, social games, and comparatively "indie" console channels like XBLA, PSN, and WiiWare.

There is also a growing crowd of capable, passionate videogame developers on the rise, eagerly looking for opportunities to create videogames for a living. Granted, pretty much any creative and skilled job in the world is and has long been highly competitive, whether we're talking about novel writing, acting, being a professional athlete, composing music, etc. However as in those fields, there will probably always be a place for people dedicated to being really amazing at it that find a way to set themselves apart

from the crowd – someone's going to be doing it, and that someone is almost without exception someone that has committed substantial time and energy into preparing for it.

11. WHAT DO YOU LIKE MOST ABOUT YOUR JOB?

It's constantly changing. This keeps it exciting!

Likewise, the outcome of every project is pretty much solely up to what I create and how I present it. With any given commercial project that I do independently, it might be hugely successful. (...but generally this is not what happens!)

12. WHAT DO YOU LIKE LEAST ABOUT YOUR JOB?

It's constantly changing. This keeps it stressful!

And again, the outcome of every project is pretty much solely up to what I create and how I present it. With any given commercial project that I do independently, it might be a total flop. (...this happens very frequently to independent developers, and is also the

outcome of many projects by developers that have had success with other titles!)

13. WHAT ARE THE SALARY RANGES, E.G. TYPICAL STARTING SALARY AND TYPICAL TOP SALARY?

The amount earned by independent videogame developers varies wildly, as has my own income from it over the years. There really is no such thing in this case as typical, starting, or top, since it depends on the success of an individual's projects in the marketplace.

14. DO YOU HAVE ANY ADVICE FOR ME AS I CONSIDER MY CAREER OPTIONS?

While I certainly encourage training yourself to develop videogames independently, I would advise thinking of that in the near term as a hobby and/or as practice for some other type of primary/fallback traditional career option (software engineer, project manager, user experience designer, animator, etc.). Properly prepared for, those jobs often pay better on average, and certainly pay more reliably, than going independent. Those positions also

create the possibility of getting additional experience and connections at an established company, hopefully saving up some money in the bank or helping to pay off student loans, all while building your credibility as a professional.

(Consider that when people find out I did technical game design work on console games at EA, even though only for 11 months and only on two different projects, potential collaborators or business partners can instantly “get” what that means with much less effort than it takes to contextualize the many years I spent before then independently making dozens of videogames they’ve never heard of.)

15. ANYTHING ELSE YOU WOULD LIKE TO TELL ME ABOUT THIS PROFESSION?

Whether or not it works out as a primary career path, I’m a big believer in there being many other valid reasons to make videogames personally, at the very least on the side as a hobby. That’s much of

what my writing is about. Besides being a fun and different way to express ourselves, it’s a context to work with skilled people that think very differently, reach strangers around the globe with our creative output, and of course to learn and practice technical skills of all sorts that have a myriad of other applications in modern life.

Should any given person try to become a professional painter, guitar player, movie producer, poet, or swimmer? It depends very much on the person, and it’s very likely to be a bumpy and very challenging path, involving a decent amount of luck on top of ability and qualifications, with significant risks along the way but potentially significant rewards. Many people can paint, but that doesn’t mean someone can automatically make a living from it; being able to simply make a videogame (or multiple videogames) independently does not automatically mean that said game(s) will earn enough money

to be someone's main source of income. However, apart from these beings done professionally, I think that the world is better off with more people painting, playing guitar, making movies, writing poetry, and swimming.

Likewise, it'd be irresponsible for me to tell anyone I don't know extremely well that they should put all their eggs in one basket and aim straight for something as unpredictable as independent videogame development as their main career plan – maybe it'll work out, and it'll be terrific if it does, but the odds are rough and it takes a particular type of person to thrive in the uncertainty and types of challenges that come with it. However I don't think that the difficulties of doing it as a sole source of income should get in anyone's way of making videogames. And in the meantime, that experience making videogames as a hobby can increase someone's core skills, community reputation, peer

network, and self-knowledge in relation to becoming a professional independent videogame developer, helping the chances that the path into it will become a somewhat more realistic and viable option.

INFLUENCE OF BUSINESS MODELS ON GAME DESIGN

When the way that games get paid for changes, the types of games that get made change in response. Business models create play contexts in which certain game genres often emerge as a most profitable fit.



This is pretty common position to hear from designers talking about the question of microtransactions:

> I don't have an ethical problem with microtransactions, except where a game is designed to manipulate people into spending extra money.

This angle imagines that money is somehow a new factor in videogame design. It isn't. I feel like we're not giving past game design a very realistic or fair view, and that some additional context here might help for the sake of perspective.

Open disclosure: I personally don't play social games, games with microtransactions, or free-to-play.

But I've certainly played, and enjoyed, plenty of videogames over the past two or three decades that "manipulated" people into spending extra money. Because I grew up playing arcade games, as well as ports of arcade games for NES and SNES. And games that were available to rent if it didn't last long enough to buy. And shareware that I wouldn't buy the full version of if I could get

enough of the gameplay out of the trial version. And games that had commercial expansion packs...

Today's game designers, since we grew up playing those kinds of games, idolize and admire that era of games. It might not seem to be so now, relative to the kinds of commercial games made now, but for starters: arcade games were undeniably designed to manipulate people into spending extra money. The ports were often one step removed from the context in which they were designed to extract quarters from us, but even then part of the appeal and positioning of ports was that here was a way to have an arcade game with unlimited quarters, a desire instilled in us by the way the games were designed in the first place. Just thinking of the game as "good" isn't sufficient in the realm of design – there are many different ways for something to be good, to different audiences, in different contexts, and those audiences and contexts were

defined into predictable channels by how they fit into the market.

Games that were two-player co-op, for example *Marble Madness* (and this is something Mark Cerny said outright at GDC a few years ago) were specifically made 2-player simultaneous so that the game could chew through twice as many coins at the same time – and the 4-player *TMNT*, *Simpsons*, and *X-Men* games clearly thrived for (and were likely developed for) the same reason. Coin-driven games are also where our beloved "Nintendo-hard" came from – the hard Nintendo games were typically ports of games that had been designed for arcades (or at least by arcade game designers), intended to kill off the player within a few minutes (in the tradition of *Donkey Kong*, *Asteroids*, *Centipede*, *Defender*, *Joust*, *Q*Bert*, *Pac-Man*, *Missile Command*, *Tempest*...).



Ninja Gaiden and the Final Fight games showed our heroes about to get buzz-sawed or blown up if we didn't continue.

Street Fighter II made the defeated fighter look beat up and shamed after the match, even with built-in (textual) trash talk.

This effect was of course amplified considerably between human players. To challenge someone else to earn back your pride in Street Fighter II then cost another coin. To practice and become become skilled enough to win when challenging someone else was going to cost even more coins. I think part of the game's success even as a console game came because when it hit SNES and Genesis arcades were still socially active (read: relevant to business) places, and the home

version gave us an edge from being able to practice without burning through change. That's what got me excited about it being a pretty faithful recreation of the arcade game when I was a kid – otherwise I certainly didn't care nearly as much then as I do now about accuracy of emulation/ports for its own sake.



Like many of the other features designed for optimization of revenue in an arcade context, these screens were kept around for the console ports, too.

In the even earlier console market on a separate design trajectory than the arcades, Atari Adventure and Pitfall hit on the point that for home games a longer play session might be desirable, and that's the direction that Zelda and Dragon Warrior / Final Fantasy later popularized even further. While it's

not nearly as common anymore due to the ability to download online demos (though I hear it technically still happens in places, and via a few mail services that I don't know anyone actually using), recall that people used to rent the full games for a couple of days at a time from Blockbuster, or Liberty, or Hollywood Video or whatever. That reinforced that games of a certain length were destined to be rented instead of bought, dictating a certain genre and length of content in order to translate to sales.

When games shifted to a shareware/episodic model in the early 1990s with id Software's Wolf3D and Doom popularizing that as a sales channel, that favored certain lengths and genres to be able to upsell people effectively to buy the full versions. Designers trying to use that model that didn't heed certain design patterns may have still gotten downloads but not sales – I personally know a ton of people

that played Snood, but no one that paid a penny for it, it was like the WinRAR of downloadable games at the time.



Briefly, expansion packs were a common way companies tried to diffuse the massive costs of developing new engines and IPs, without going to the full length of generating expensive sequels with higher expectations. We've seen a shift away from that toward more granular DLC, but the same kinds of design and production discussions are still happening, just about, "What kind of game can we make for which DLC will work well" instead of about genres that players would buy an expansion pack for.

Steven Kent, in *The Ultimate History of Video Games* [sic], explained that Pokemon – an IP that a whole generation adores – was an idea executed by Nintendo because it would get Game Boy players to buy and use Gamelink cables. (I.e. in case the theme being “Gotta catch’em all” didn’t make it sufficiently clear, the Pokemon franchise was explicitly designed to manipulate people into spending extra money.)

The goal of MMO’s is to keep people tied up enough in continually added content and their social obligations that they won’t cancel their subscriptions. The goal of addictive little mobile games is to get people to keep pulling up the game frequently enough when standing in line or waiting for the bus or before class to get others around them to notice them playing and ask about it (I’m looking at you, Angry Birds).

The method of a game recouping its costs dictates certain priorities in its design. Games that

capitalize on a channels affordances become the highest grossing in their category, making them the most recognized brands and the designs most copied by peer companies in an effort to split that pie.

When I first heard anything about microtransactions in the United States, before the iPhones or Facebook apps, it was from a (at the time fellow) AAA developer telling me that someone had finally found a way to get China to pay money for software. The Chinese players would pirate full games, there was simply no way to sell them \$60 USD retail, that’s not how that part of the world works/ worked. However they would willingly throw down a few pennies here, a nickel there, even a dime to get temporary boosts and benefits in their games. Bam, just like that, a billion more potential customers just fell within the industry’s sights. Not long after that, developers discovered that character customization

microtransactions were often more palatable to Western gamers that were typically more obsessed with meritocracy and expression of individuality.

When we get into games as players, we don't necessarily think about the connection between the game's design and how that connects to its method of payment. We just play the games we like. When the dominant economic models that games get paid for change, then the design principles that are successful within those markets change, and we may find that our tastes were left behind or become niche, while products we don't care for suddenly become the primary interest of new companies making the next wave of games.

I know some people that really, truly, genuinely enjoy playing social games on Facebook, that feel like microtransactions are a perfectly acceptable and normal part of play. They don't like the kind of games I like. I prefer the

old arcade game level of difficulty – I turn to pinball machines to feed my antiquated tastes (as much designed to manipulate people out of their quarters if anything was) – and I realize that doesn't suit everyone's tastes.

I acknowledge though that the games I grew up playing were often designed with the economic models of their time in mind, and that the same is true now, only the methods and timing and nature of payments have just changed. The types of games designed to seek profits in these new and different conditions and payment schemes haven't necessarily been my cup of tea, but they clearly appeal to a lot of other people.

It seems almost as though the metaquestion actually being handled in these types of discussions is something more along the lines of: what are players who got into games for how they used to be designed (as a function of formerly dominant payment

schemes) ok with in games that rely on microtransactions?

To the extent that many players of microtransaction games – I even suspect most players playing those games – weren't and aren't actually players of "core gaming" releases on HD consoles, arcades, or top-of-the-line PC rigs, but were instead maybe the sort of player who liked Peggle and Diner Dash and WiiSports or computer MahJongg, it seems like this line of discussion risks skewing our priorities as designers even further toward satisfying our own interests based on the types of games that we grew up playing. We're probably not a particularly representative sample of the players that microtransaction games seem to genuinely appeal to.

WHAT'S JAPAN DOING DIFFERENTLY?

Though Atari, a US company, helped grow the industry, after 1983's crash Nintendo resuscitated the industry to the form I saw as a kid. I've long wondered how Japan's game making might differ. In 2006 I got a few glimpses.



In 2006, I had the opportunity to sit down for lunch with Takayoshi Sato, the CGI Director of Konami's Silent Hill (1 and 2). I approached him with an interest in learning more about how videogame business and design happens in Japan, which has historically produced dramatically different work than we have seen from the American process.

I wish to emphasize: this informal exchange took place many years ago. It may or may not be representative of Takayoshi Sato's

current views. Projections and theories presented here may or may not be representative the Japanese videogame market today. As a point of reference: the Wii, PS3, and iPhone had not yet come out, and World of Warcraft was only 2 years old – this industry can change a lot in a few years!

Though I did not record the interview verbatim, I did take detailed notes. Although I considered reformatting the notes into a more conventional interview

format, ultimately it would have included the same information, but it could have also invited the risk of quotes appearing “from”

Takayoshi Sato in my words. The notes here are my notes, in my wording, from the conversation. Although my goal in this transcription has been to convey this developer’s message, my efforts to flesh out my notes into a full sentences has pulled in a few details from what feels like clear memory, and my notes at the time also likely included a measure of interpretation or deduction.

My notes from the informal interview

- American developers engage in internal sales pitches, flashy demo projects, detailed proofs of concept, early visual renders and such to battle for budget increases. In his experience, this didn’t happen in Japan. Teams are assigned realistic and practical budgets for their projects, and then the teams stay within those budgets.

- The core of project planning begins around, and encircles throughout all development: player control, player movement, and camera behavior. Gameplay mechanics, enemy/level design, artwork, and programming mostly surround and support those systems.

- Mark Cerny (Marble Madness, Sonic 2, Crash Bandicoot, Jak and Daxter, Spyro the Dragon, Ratchet and Clank) started in the US with Atari, but spent years as a developer in Japan, and so is an example of someone who has experience and knowledge from both sides of the Pacific. His most influential contribution to industry practice was popularizing the “Cerny Method”, in which a project doesn’t “go wide” into making all characters and levels until one representative level is fully playable, through several design iterations, and decided to warrant investment into full production.

- A lot of videogame development in Japan currently seems to be shifting to mobile, but there's no telling what the market will look like in the future.

- Japan tends to hire people right out of college. Although something with a business or software engineering focus may be sought for certain types of positions within the company, there are not yet videogame-specific degrees forming any considerable part of the Japanese videogame industry.

- Japanese companies tend to keep employees for most of their working life. This is done partly as a means of preserving company secrets, partly as a protection of valuable employee relationships, and partly because the culture regards leaving employees as betrayers.

- In Japanese companies, there may be many workers at any given time sitting around not contributing on a given project or

during a given phase – but it's understood and respected by their peers that those people have given more of themselves in some other projects, and will likely find their way back into future project involvement when the time is right. This is in stark contrast to the frequent turnover in American videogame development companies, where employees often get restless or employers often change direction sharply every couple of years.

- There are few, if any, significant books or courses on videogame development in Japan yet. They're likely to appear in the not-so-distant future, but due to the secrecy of business in Japanese companies they're unlikely to reflect industry practices.

- Nintendo is a particularly big question mark to everyone. Virtually no one that works first party there ever leaves. Nintendo is known for paying notably high salaries than many other studios.

- Japanese videogame studios are often merging, and mostly shrinking.

- Capcom and several other Japanese studios (possibly including Namco?) closed their American studios. Nintendo is one of the few with a foot still firmly in the US.

- There are virtually no studios in Japan with positions for “game designers” in a non-technical sense. The director, programmer/designer roles, and artist/designer roles work together to cover the job function sometimes relegated to “game designers” in some US studios.

- Japanese videogame artists work almost exclusively in the resolution, format, and fidelity required by the game, concerned almost exclusively with how their art looks during play. Concept art, and the sort of high detail imaginative sketching that appears on packaging and manuals, is often produced after

the videogame’s art needs are filled, while waiting for the game to ship. Rather than being a process step – except in the case of very rough sketches for personal planning – this sort of art gets done late in the process separately, explicitly for use in marketing materials.

- Creative people in the development process in American studios sometimes become embittered over feeling lack of control in the work they do, losing a sense of ownership. Partly from the culture in Japan being one that emphasizes duty and mutual respect, creative people in Japanese companies seem to be better able to take direction while still feeling an immense amount of ownership in their work.

- Everyone on a Japanese videogame development team generally understands or sees most of the big picture. There is not an intensely specialized attitude of just being with the company to work on an isolated

part, and everyone shares interest in how the work and final product will come together.

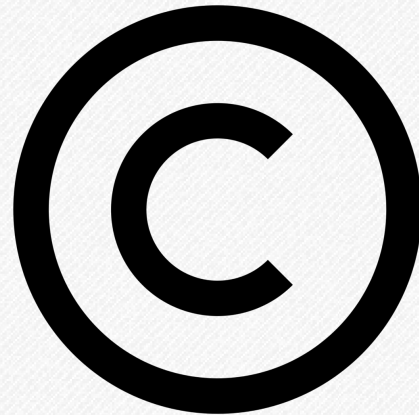
- The culture of duty and mutual respect also results in games which have a much more focused creative vision. Although most people on the team have visibility on and interest in the big picture, they work toward their common goals without the need to force their own ideas into it. This may also be a side effect of the tendency for much lower job turnover – in the US, everyone feels like the features and visible pieces they work on are critical to landing them their next job or role in another company; in Japan, most developers aren't desperately trying to prove their individual ability because they're at home with their employer, and feel the most pride from having been a part of the team that worked together to realize a successful vision.

- No company, on either side of the ocean, is so big or so powerful

that anyone can be certain it'll still be around 5-10 years from now. History is full of surprising upsets that shook up huge companies and industries practically overnight.

INTELLECTUAL PROPERTY AND HOBBY GAME DEVELOPMENT

On the one hand, I.P. law is an intimidating and complex field, in many ways best left to lawyers. On the other, any time we make anything I.P.'s automatically involved! Here's a starting point for what to look into further.



Q: I was thinking about making a game as a hobby based on a licensed property. If I were to release it, I would not charge for it and it would be free to anyone who wants it. Would that still be an issue if I don't secure the rights to use the property? (BTW, the series is... pretty old at this point, so I'm not sure that even if I didn't have the rights to do so, if the owner would even pursue any action.)

A: Hello! As always, I want to thank you for taking the time to write. It's always helpful to receive questions, since whatever the question is, as with in a classroom you're never the only person wondering about it.

I Am Not a Lawyer

Necessary disclaimer: this is not intended to serve as nor be a substitute for real legal advice. The responsible and best thing to do to stay out of trouble is to consult a lawyer with specific questions before acting on such matters. If you do decide to act without a lawyer's guidance on these kinds of matters, that would purely be your risk to bear, not mine, because as mentioned, this really is not intended to serve as

legal advice or substitute for that kind of thing.

(Void where prohibited. Some assembly required. Not available in all areas.)

But I Am a Game Developer

I'm just another non-lawyer trying to offer another perspective from my own experience making games and wondering about, reading about, and maybe even sometimes coming a little uncomfortably close as a practicing videogame developer to crossing those same kinds of lines in terms of IP.

This is going to be a little overly thorough, since I'm adapting this to be a more general entry for more than just the reader who wrote the question. So I'm going to begin by giving sort of a high level review of some of the main areas of intellectual property, as they might affect our videogame development.

TL;DR Short Answer

But first of all, to get to the short answer...

If I had the venture a guess of the technical answer: it can still absolutely be an issue, and therefore no, you really should not do it.

But life is a lot more colorful than technical answers tend to reveal. At least from a practical standpoint, there may be some gray area here, or at the very least, some sliders along a continuum to consider when thinking about what you're comfortable with and what remains true enough to the purposes you have in mind.

Main Terms: 4.5 Kinds of IP

We'll begin with a primer on what's a short take* on "4.5" main categories of IP. This won't seem super short, but as a reminder: shelves can be filled with countless thick volumes on details and particular cases about any one of these, there's people who specialize in these topics and

devote years of their lives just to work with it. This really is the short version.

1-TRADE SECRETS

Company processes or recipes that aren't public knowledge, which help give a company a competitive edge based on what they know that they take measures to prevent competitors from becoming aware of. I have the least familiarity with this one, but loosely speaking I believe the gist is that if someone finds out exactly how Coca-Cola is made, and then sells that information to a competitor, it could have a negative financial impact on Coca-Cola, the competitor could get in trouble for using it, and that someone could be in even bigger trouble for his or her role in the exchange.

For a videogames example of this: a few years ago when social games were still massively on the rise, new competitors entering the space against Zynga were hiring away employees from Zynga, at

every level from executives down to entry-level line workers. Some of people coming from Zynga, or so some reports seemed to indicate on the web, were potentially taking with them documents about Zynga's game development and monetization processes, which is of course no good. That's fundamentally an intellectual property issue, though about trade secrets, which really isn't closely tied to the question you've posed.

2-PATENTS

Patents give inventor's rights to control who can manufacture and sell their inventions, in exchange for the details of that invention going on public record and becoming public property after some period of time.

Patents most often are associated with physical inventions, like a more efficient and reliable mechanism to pull a roller coaster up rails, but sometimes intersect videogames when they relate to proprietary hardware (I believe

Nintendo has patents on how their classic D-Pad works, which is part of why many competing D-Pads felt/feel flimsy or inferior), complex methods/algorithms (John Carmack figured out a self-shadow algorithm for Doom 3's lighting engine he wasn't able to keep in the game because some other company had a patent on it already), or occasionally even weird gameplay features like Namco's patents on playing minigames during load screens or ways of handling fighting move tutorials in-game. Patents are crazy expensive, but can sometimes be worth money when a company gets acquired, which is part of why people invest in them, and also get justified as a form of corporate mutually assured destruction, i.e. if your company comes after our company over complaints about your patents we'll come after you with ours, and so they both want to have an arsenal built up of their own and from acquired companies. I

believe just this sort of thing exploded in the case of iOS and Android devices, but not really our topic here. Note too that a patent can be filed but then turn out to not hold up in court, just because it exists doesn't necessarily mean it's valid, which is confusing and weird and slightly terrifying, and why lawyers get paid a lot of money to sort that mess out.

Patents in software are a lot less common than trademark or copyright considerations though so we'll shift that direction. I wanted to acknowledge those as categories though, to distinguish it from these other two, which are more closely related, without misrepresenting these next two as the whole of what IP seems to be about:

3-TRADEMARK

Trademark is a word/phrase, stylized logo, or even a fictional character/mascot. If your game has "Star Wars" in the title, if you use the PS3 logo without permission, or want to include a

depiction of the Geico Gecko on your game, those could fall under trademark infringement rather than copyright infringement. These get protection for a different reason than patents, here it's because a company or individual invests money into creating brand value, recognition, and associations with their label, which then has sellable worth, and if others go use it without permission it could dilute its value by, for example, Marvel gradually losing authority over the public representation of their X-Men superheros, or in another direction devaluing people's perception of Marvel as a brand by associating it with junk people put that label on.

This is also to protect consumers from deliberate brand confusion, rewarding companies that take responsibility for both quality and mishaps. So if Sony invests a ton of money into establishing its reputation as a manufacturer of reliable consumer electronics, then some crummy knockoff company

that cuts corners on parts and processes shouldn't be able to slap Sony on their alarm clock to sell more clocks. They'd be benefitting from Sony's value by misleading consumers about what they're buying and from whom.

4-COPYRIGHT

Copyright is (loosely) protection over a specific implementation of an idea, and is focused largely on creative, artistic artifacts. Whereas patents can be extremely expensive and complicated to file, and trademarks can be fairly expensive but comparatively simpler in relation to, say, patents, with copyright thanks to the widely-accepted Berne Convention you basically have automatic copyright on every version of everything creative you do at the moment you do it. (Note that if certain work is being created for pay, part of the agreement signed may explicitly transfer the copyright or other types of IP for that specific work to the paying company.)

Everything you write, paint, draw, play on piano, etc. you have a copyright over, and that encompasses a set of rights, any one of which could be sold or granted independently of one another, such as your right to be credited for its reuse, whether someone's allowed to reuse it at all, whether someone is allowed to make variations of it, whether someone can sell it or a variation of it...

Also note that even though copyright is automatic, if it comes down to a court case paperwork and some modest filing fee that needs to take place for that to proceed. If that registration paperwork fully existed well before the complaint began I suspect (again, not a lawyer!) it probably makes for a stronger case. It's also suggested to have a clearly visible copyright notice ("Copyright 2014 by Chris DeLeon" or "© 2014 Chris DeLeon"), to be able to later make clear that you've claimed the copyright.

4.5-PERSONALITY RIGHTS

The last 0.5 of the 4.5:

"Personality rights" relates to the rights to use a famous person's likeness, name, or possibly real voice in association with a product. I'm not sure whether they are more related to trademark, copyright, or something else entirely, but they're absolutely a real thing. Sometimes a game based on a movie has a generic hero face or generic voice actor instead of the real Hollywood star not for lack of a good reference photo or capability to extract usable vocal audio from the movie, but because celebrity likeness rights can be expensive on top of and separate from the actual show or movie license.

Breakdown

So far sake of example, at least as I understand it:

If you put the show's logo on your title screen, or the name of the show in your game's title: trademark issue.

If you took the show's theme song and played it in your game, or a clip of video from it, or even a model/texture/sound from some other company's game: copyright violation.

If you wanted to show a depiction of the main actor or actress: personality rights concern.

Fan Works

Fan fiction and related areas of fandom by the way is a terribly complicated legal territory, and I suspect with a lot of case-by-case variation between company attitudes, and a great deal of consciously looking the other way.

Back when it started as a cultural phenomenon, huge companies seemed to have a scorched earth policy about shutting it down everywhere it cropped up, in order to maintain total control over their IP. But in the past several decades some IP owners have opted to embrace fandom and fan culture and fan fiction as a part of a healthy fan community. They

might go as far as hosting conferences, embracing/ highlighting the best of it, maybe even offering some official involvement (like a company hiring a modder to join the real development team). But that is by no means universal and should never be assumed.

What I'd Consider

Now, for real, back to the actual question, with a focus on the practical considerations I'd be weighing:

NOT CHARGING DOES NOT MEAN FAIR-USE

1. People on the internet often spread the impression that as long as you're not selling it, it's legal and fair to do. At least as I understand it, that's simply not actually true. Practically speaking, it probably does paint a much bigger target on your back if you make millions of dollars misusing someone else's IP, and that company might chase after you for its cut and/or to shut you down for doing that. Practically speaking, even though huge companies have entire departments full of lawyers, they're pretty busy with a lot of real business to worry about, and whatever someone's doing without making any money and especially if it's not drawing any real attention is probably a lot less likely to get them to make time to come after you.

But it also happens sometimes that some teenager somewhere is just making a mod about their favorite childhood IP, then a lawyer sends them a generic/form, a Cease & Desists letter, that says either shut that down immediately or face further legal action. The C&D letter takes them no longer to create than it does to type an address for an envelope and add postage, and with the receiving party knowingly on empty legal ground for what they're doing (having gotten no permission, no licensing, plus usually no real financial or legal resources to fight their seemingly defenseless position in court anyhow), this generally means instant death of that project, which is always a real risk when toying with IP that isn't theirs. Note by the way that as a general rule of thumb (again: not a lawyer, not a lawyer, not a lawyer, I might be totally wrong about this!) provided you're not making real money off it seems like a lot of the time you're more likely to get shut

down by an intimidating letter than fined an exorbitant amount or thrown in jail. I'm sure there's exceptions to that, but that seems to be at least in the handful of cases I'm familiar with, what often goes on.

Unless of course you're messing with the record industry, as their policy seems to be to "make an example of" people by randomly selecting targets who once had 3 mp3 files on their computer then putting them into debt for decades. But record industry isn't really what we're talking about here.

Many people think that what they're doing is "fair-use" when it absolutely isn't. There are certain limited uses for the purposes of art, education, commentary, criticism, parody, news – but there are a ton of details about those particulars written up on the web by actual lawyers if you're curious. It doesn't actually matter if it's "pretty old" unless it's so extremely old that

falls into public domain. It doesn't actually matter if the original owner is no longer making money from it, or even still in business, because the rights to decide when/how/whether to begin generating revenue from those properties again get passed around and sold as business assets for long after their initial use.

Don't trust my word on it or on web forums, before deciding that what you're doing is Fair Use. Really, you should be digging for the explanation of Fair Use by someone who has really committed the time and education to learn the ins and outs of it. Of course even if it's a website or article unmistakably written by a lawyer it's still just a starting point, and a real conversation with a lawyer could be instrumental in staying out of trouble from a misunderstanding.

Plus, an unsettling secret I've run into from working with lawyers: not all lawyers see eye to eye on

exactly what some of these things and their ramifications are, either! Sorting out what's right in any given case is part of why we have a debate-like court system. What your lawyer thought can turn out to be wrong.

Even in the event that you're right about something though, proving that's the case in court can be time consuming, costly, and a ton of hassle, so unless you've got a really important artistic or critical statement worth raising a ruckus over, or the strength of solid representation (in which case I have to imagine you wouldn't be e-mailing a game designer a law question) it's probably preferable to just stay out of their sights of these kind of people and not risk drawing any more attention than you have to.

WHO WILL SEE IT?

2. *Who will see it? If it's you and your roommates, you can probably get away with virtually anything. Although even if it's you and a classroom, that's still probably true, but what if one of them records it with their phone and posts it to YouTube, and then what if it blows up*

getting a ton of views, shows up on the front page of Reddit, and the company gets wind of it that way? Now, it's unlikely, but it's possible, and if it's not something you'd be comfortable with, that's something to consider.

Now I've admittedly made some small throwaway noncommercial videogame projects designed specifically to reach tiny audiences, measuring in the hundreds at the most, that I probably came closer to using IP in in ways that I absolutely would never do in one of my "real" games, either commercial or even purely as a hobby, that had potential for significant reach and visibility.

SIMILAR, OR INSPIRED BY, NOT LICENSED

3. Is there a way that you can be safer or more clever about it by speaking to the same audience, even appealing to some of the same themes or ideas, without going so far as to incorporate any of the original characters, music, title/logo, etc.?

Space Invaders wasn't originally going to be about space conflict, but it changed in development when the developer saw a Star Wars ad. A ton of films and games around that time were capitalizing

on widespread interest in space from the moon landing.

Around the time that Fast and the Furious and Gone in 60 Seconds were big from theaters, Need for Speed Underground appeared with options to upgrade neon under lighting and custom mufflers on street racing cars. The same year District 9 came out in theaters a game about battling space aliens came out called Sektion 8. The old Contra games heroes and enemy characters were blatantly inspired by movies like Aliens and Predator. The pinball machine F-14 Tomcat came out shortly after the movie Top Gun prominently popularized that model of fighter jet. After The Matrix popularized bullet time Max Payne capitalized on it with time-slowing pills. The spiral effect on the rail gun in Quake 2 was based on the effect for the rail gun from the movie Eraser, and the see-through-walls effect of the Farsight in Perfect Dark was based on the functionality of that

same gun from Eraser (the MagSec in Perfect Dark also closely resembles the sound and appearance of RoboCop's pistol). I imagine it goes without saying that after Twilight became huge, we were seeing werewolves and teenage vampires everywhere, and that Harry Potter brought in its wake a ton of other kids books about little witches, little warlocks, and little wizards.

StarCraft, depending who you talk to, seems to be very blatantly an appeal to fans of (or arguably a rip off of) Aliens, Predators, and/or WarHammer 40k.

So can you make a game based on a licensed property without securing the rights? No. Not really, not technically. I mean you're probably capable of doing so, however the better your game does, even if it's free, the more risk you've put yourself in for trouble. But what you are far more likely to be fine doing, at least based on observations of what so many other high visibility companies

seem to have gotten away with doing for decades now, is to borrow or build upon the basic idea, as long as you riff on it, find a way to really make it your own, and don't try to imply any association official or otherwise with the property in question.

Probably ok: a game about teens that slay vampires.

Probably not ok: a game called Buffy the Vampire Slayer, featuring Buffy the Vampire Slayer.

Somewhere fuzzy in-between, and exists on a sliding continuum from cowardly to really playing with fire: a game about teens that slay vampires that has clear allusions to or similarities to Buffy the Vampire Slayer but doesn't specifically call itself that or otherwise make the association explicit.

Anyhow, that's the best of what I've got to offer on the question at the moment. Hopefully it's at least a little bit helpful in providing some additional things to consider.

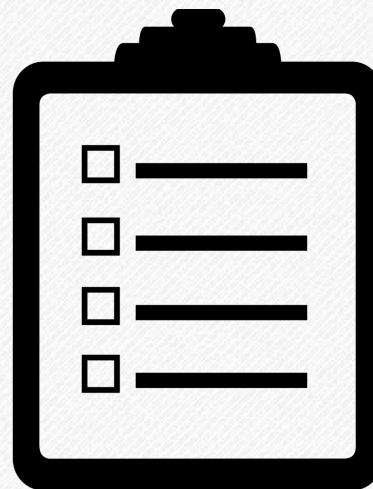
Whatever you decide to do, good luck!

Talk to an Actual Lawyer

And again, it's really important to stay out of trouble in these kinds of things. If you've got the resources I really recommend talking to a lawyer about any questions you might have. Or, at least, find another source that's not just another game designer trying to talk as best he or she can from his or her own limited experiences and reading.

SHOULD I RELEASE A GAME AS SOON AS IT'S DONE?

Many developers learn the hard way - a few times over - that if your goal is to get people playing your game, just finishing the creation of it is only one small part of many to that objective. You'll want a launch plan!



The question sounds pretty obvious. Of course, right? I mean c'mon, it's done!

But the answer is no.

Releasing a game just as soon as it's done is kind of a bad idea.

When someone is new to game development, and they're mostly just making projects as way to figuring out how things work, it's not really a big deal. Once someone has a bit of experience though, and they've invested themselves into crafting

something that has potential to be of interest to strangers (a good litmus test, since strangers won't just play it to be nice to us), there are at least two more small steps to take after completion before releasing it.

First, if you haven't already done so earlier in development with a build that looked finished, it may be useful to promote the game a bit and create some decent promotional materials to point people to. A short and simple YouTube video will do, but make it

entertaining instead of descriptive. Also, take a bunch of screenshots to pick out the few best to represent the game on the web. Maybe even gather a few player testimonials from private testing.

Lots of people either don't play games or don't have the time or set up to play your game right this minute. That includes classmates and family members on Facebook, friends without the particular platform or specs you develop for – even recruiters at game development companies (you'd be shocked by how many people working at game companies don't play videogames). Having a decent video can help these people quickly learn about the game, and even share it with others that may be able to play it.

Meanwhile for people that do have the time and system specs to play your game, watching even a half minute of the developer showing one way to play the game the way that it was intended (even if not the only way to play!) can help

players get started quickly instead of fumbling around trying to figure out what's going on.

People are busy and there are a ton of exciting, free, easy-to-reach things competing for everyone's limited free time. Watching a sweet video may be just the thing needed to bridge the gap and convince someone to try it. If they decide it isn't for them, no big deal, but at least then they can base that decision on information other than the game's title.

Retail commercial games come in a packaging that allows undecided players to get a quick sense of the game by the cover art and information+screenshots on the back. Think of the video, screenshots, and (if relevant) testimonials as your version of packaging. And just like it would not be acceptable for a game company to just sell their retail games as unpackaged DVDs laying on the shelf, even otherwise happy players will pass on your

game if you don't show enough interest to package it decently.

The second thing, and this of course comes before packaging but it's equally important, is to really user test the 'final' game with people that have never seen you play it or had you explain it. This isn't really about fixing bugs, so it doesn't matter how thorough of a programmer you are, it's about identifying accidental design slips that you would likely apologize over if you saw someone struggling with it. Yes, it's necessary to write a game that doesn't crash, and it's necessary to ensure that it installs and/or runs decently on a variety of machines, but I'm assuming that you've solved those problems already.

This is about having one last chance to smooth over, fix, or improve the places that new players get stuck or confused by when sat in front of the game without help. If someone needs you as the developer to explain,

clarify, coach, or point something out to them when they sit down to play the game, multiply that problem by a thousand players and realize that you won't get an opportunity to talk to any of them.

Take whatever it is that you find yourself desperately wanting to tell your playtesters and either embed that information blatantly into the game at a relevant time. Or, even better, if at all possible find some clever way to remove the need to provide that clarification. You will always, always catch something surprisingly important by doing this for each game. Not doing it would be like shipping code without even first trying to compile it to see if any errors come up. Madness!

Testing the would-be-final version with a few people can make a big difference. It's well worth putting a few extra days into, at least. If you don't feel like you can even bother a few of your personal connections to try it, are you really

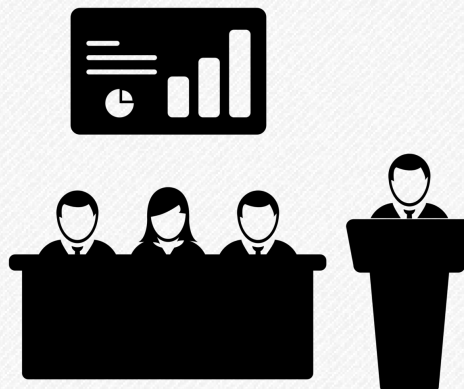
sure that you're ready to bother a few thousand strangers with it?

By comparison to how long is needed to take a game from concept to release, we're really only talking here about a tiny percentage of total development time – maybe a few percent for some immediate gains, or a bit more if you're really wanting to polish it, say, for commercial release.

That little bit of extra work at the end, even though the game is technically functional enough and feature complete so it could be released, can greatly amplify the size and satisfaction of your audience. While there's obviously more to being happy with a game project than simply hitting bigger numbers, these two practices can also help your game wind up in front of the sort of players that it's intended for, and increase the probability of them seeing all or more of the content that you (and your team, if applicable) worked so hard to create for them.

HOW TO GET THE MOST OUT OF YOUR FIRST GDC

The GDC - Game Developers Conference - is one of the largest conferences in the world for developers. For people interested in the possibility of transitioning from hobby to professional, it's well worth looking into.



WHAT IS THE GDC?

The question at hand today is: how do you get the most out of your first Game Developers Conference?

First, for readers less familiar, a bit of background: the GDC happens every year, almost always in San Francisco. It used to be in San Jose sometimes, now it's pretty exclusively in the middle of SF in the Moscone Center. People come from all over to attend. It's a developer-focused event. Oftentimes students will go out

there to visit while they're still in school, shortly after graduation, maybe even a year or two out as part of their job search and professional development. It's part of how they're trying to build their professional and peer network.

I've been to all but two of the GDCs since 2005, so I've seen more than a handful of these. On most of those visits I've got other people coming for their first time who were in one of my game development clubs, back when I was at Carnegie or now that I'm at

Georgia Tech. So I've got some practice at trying to help those people get the most that they can out of that time. I'm going to pass on to you some of the advice that I think has helped reap some rewards for those folks, including myself (these are things I still do every time I go).

BRING BUSINESS CARDS

At the top of that list is to make sure you make some up-to-date business cards. Make sure it's got your name, e-mail, cell, a website for your portfolio or games, and a few words describing what you do. If applicable, a few highlights from your background. I like to think of mine as a mini-resume, it'll help people remember me and what I do. I don't care actually if you already have a professional looking card leftover from the last company you worked for, or the school you're attending, people probably aren't asking you about the past company you worked for or your school, they're asking you about you. You should list that

company or list that school on your personalized card, but you should also make sure you mention whatever differentiates you and your work and your strengths and your interests. It's a chance to highlight what'll separate you from the other 500 people that they talk to that day. And also, if they're trying to find you in the shuffle after-the-fact to contact you for something, they need to have some sort of hooks on your card to whatever it was you were talking about to get you back in touch with the right person. Because one of the nice things about doing this is that people can build new connections this way. I encourage keeping the backside blank – it's cheaper, and also makes it easier to jot down notes on the back (you should carry a couple of pens!). Sometimes the person you're talking to, they're out of business cards, they want to give you their info, but they're out of cards. That happens at GDC because they're

giving so many away. If you bring pens and you've got blank backs on your cards – this happens to me all the time – they can jot their info on the back of one of your cards, and you've still got their info.

USE YOUR PASS LANYARD

Keep a small stack of those cards stashed in the backside of the lanyard that holds your conference pass. There's actually kind of a pocket hidden for them right there. That way they're on quickdraw so you're not rooting around in your pockets or digging around your bag trying to get bent cards out.

While we're at it, the lanyard usually hangs too long, and people really don't want to look at your crotch to figure out your name. Tie a little knot behind your neck to shorten it so it hangs at a more reasonable height in front of you, and more people will use your name when talking to you.

MAKING CUSTOM BUSINESS CARDS CHEAPLY

Now there are a million ways to order business cards cheap, or

even for free on the internet. But what I like to do instead is to make a simple design in Adobe Illustrator (make sure they'll be standard card size when printed, whatever that is, I think it's roughly 2×3 inches but double check that), lay out 4×2 (or 5×2, whatever fits on a page) edge-to-edge for a full page layout. You'll get 8 or 10 per page, go to a Kinko's, print them out on white card stock – the right kind of card stock for business cards – then use the paper cutter there to separate them all yourself. I do my own. It's a lot cheaper, you can make a ton of them this way, and it only takes about half an hour of simple repetitive labor of chopping some pages. If you bring your own design there, print it out on their printer, on the nice card stock paper, you'll get a higher quality than if you tried to do it at home, without all their insane costs that they would charge for designing the card or cutting the card etc.

BRING A LOT OF BUSINESS CARDS

And by bring some business cards, I mean make and bring hundreds of business cards. 200-300, upper bound 500 wouldn't kill you. Give them to almost everyone that you talk to. Typically they'll give you one back. Don't be stingy with business cards. They aren't just there for recruiters, they're for the people you run into, because maybe they'll run into someone that they'd like to put you in touch with, and to do that they'll need your information. It's an easy and appropriate way to segue a friendly but not very professional conversation toward the direction of, "oh what are you working on," "what do you do" conversations just by giving someone your card. It's also a really easy ice breaker at conferences to swap cards, because there in each other's hands you've got something to talk about already, even if otherwise you're not very natural at spinning up a conversation with a stranger.

Everyone's there to network, and they get that. When you give them a card you're speaking their language.

AND UPDATED RESUMES

Likewise, print and bring enough resumes that you won't find yourself being stingy with them. Dozens, which is probably ample, but err in favor of not needing as many as you print, it's good to have more than you need. Contexts for handing over resumes are much less frequent than business cards, but if you find yourself ever not handing someone a resume because you want to make sure you have enough later in case you run into a better connection, you're losing opportunities, giving up opportunities, and for the stupid reason of having not carried enough sheets of paper. Don't let that happen to you.

PRESENT YOURSELF PROFESSIONALLY

Speaking of which, I assume this should go without saying, but since it is advice for the first-time

GDC goer: update your resume, pay attention not just to the information on it but also to how it looks aesthetically when you print it out. Print it out, put it in front of a friend, get their feedback. Often another set of eyes can spot issues you thought were fine, it doesn't take any special training. Then you can fix those things and put it in front of another friend, and keep doing that until they stop giving you corrections to make to it. Print it out on decent resume-quality paper, not on ordinary lightweight printer paper.

I know that can sound silly, to not just focus on the information printed on the page, because it should be the skills that count, or your education, or something else. But HR people and business people, the ones that are in positions to make hiring decisions, typically have a business background, and people with a business background look for little signs that the person they're talking to is socially aware and

adult enough to operate in a workplace with professionals of different disciplines, backgrounds, and varying levels of seniority. It's a signal to them, so don't signal to them that you're careless, unaware, or unprepared by having resumes that are clearly in first draft state or that are printed on standard paper.

Likewise, and this is probably a given for most folks but there's always a few people I wish someone had said something to: while a tie would be out of place – unless you're shooting for a business position – do your best to clean up a bit. Get a haircut before going, and wear some reasonably nice-ish clothes rather than something silly that will stand out in an unprofessional way. You'll see these people there. It's unfortunate. If you or anyone you know was going to be those people, please reconsider, it's really not doing anyone any favors in terms of landing jobs or making real connections with other

professionals, to dress all goofy. It's not Comic*Con, it's a professional conference for developers who are networking for professional reasons.

FIND OR SET UP LUNCH, EVENING MEET UPS

Make it your goal during the day to find something to do after sessions close for the night. Identify a group to tag along with or meet up with at the end of the day, or at least find a party that you can get into, or make a friend that you'll know well enough to recognize if you run into them the next day. Be there first and foremost for the people, not for the booths or talks (except to the extent that specific people are at the booths and talks). That's where the good things happen, that's where jobs happen, or where life-altering introductions take place, or great idea sharing will happen – it's the people, not the booths or the talks. The best networking during GDC really happens outside of GDC in the evenings, but you've got to set

that up during the day for it to work.

Try not to eat alone if you can avoid it. That includes lunch. Round up some people nearby or find a group nearby and ask if you can join them. It's a professional conference, they get what you're doing, and if they say no, who cares, you may never see those people anyway. Go ask someone else. Ask them about what they do, what they're hoping to get out of attending the conference this year, and introduce yourself too of course. Worst-case scenario, even if they say no they'll remember you as that kid who unknowingly had the guts to ask a bunch of famous developers you didn't even recognize if you could join them for lunch.

And sometimes that can totally go the other way. True story: at my first GDC I got to eat lunch with David Perry – the guy who invented Earthworm Jim and MDK – because it didn't even occur to me that maybe as a sophomore in

college I shouldn't have asked him if I could hang out with him over lunch. But I did. He said fine. I got to chat with him about Earthworm Jim. That was pretty sweet.

LISTEN TO PEOPLE, HELP CONNECT THEM

One of the questions that came up was, "How should someone looking to get a job in the industry make a positive impression?"

I think the first thing you can do is introduce people to each other. Be that guy or girl that people will thank later for the connection you helped them make. Run into a sound guy who wants to get experience? Play some beginning indie game that clearly could use some sound help? Get those two people in touch (hopefully you'll have their business cards to make that easier to do). Tagging along with an animator that you just met? She showed you some of her work on her laptop? When you two introduce yourselves to other small groups, tell the other people around you about the animator's work you just saw. Help propel

their connections and their careers. Oftentimes people return the favor. This is a surprisingly easy to do, with the number of people that you'll be running into, many of whom are really doing cool stuff. This is of course especially easy if you came prepared to card swap with pretty much every group you run into.

DON'T GO IN THERE "COLD"

Really prepare to talk about things that people there will care about. I don't mean industry gossip or about upcoming releases. If it's treated as a gamers convention it's really not worth the cost of attendance and many developers will find that immature or annoying anyhow. Practice talking about the highlights of your work – not all the details, just the short pitch. If you haven't read anything about videogame development, design, or the industry in awhile? Read something, anything, whether it's some trending blog posts, a book about game design, maybe my site, an academic paper if that's

more your crowd, whatever. But be ready to have some ideas to bring to conversations that aren't your own [citing/crediting the origin of those ideas, of course!]. Besides coming across as someone that's knowledgeable and passionate about what you do, you'll have some fresh ideas in your head ready to be combined with conversations and other inspirations floating around at the conference, and sometimes that'll lead to something good.

Plus one of the cool things about GDC is that you may run into the person who wrote the thing you read. The book, the article, the paper, the blog entry. You can have a conversation about it, they'll be happy you read whatever it is that they made.

PASSES: SMALL (EXPO), MEDIUM (TUTORIALS), OR LARGE (MAIN)?

The other question that came up was: "Is it worth paying extra to attend the talks and access the paid section of GDC Vault?"

I tend to compromise, I tend to go with a tutorials pass. It's much cheaper than a main pass but it puts you there for 5 days instead of 3. Being there for 5 days means a lot more networking than being there for 3 at the expo. Plus at those first two days of the tutorials you get immersed in groups with other people in whatever tracks or fields you choose, and that's a great way to build a foundation into roaming the expo floor and talking with people for the next few days. If you can only make it to the 3 day expo due to travel costs, ticket costs, or time, you can definitely still make that count so don't be disheartened, but if you can swing a tutorial pass I've always found the extra two days of field-specific networking totally worth it.

But to be honest I really question the value of the main talks passes. They're super expensive. Typically it's some talking head in the front of the room, while you don't really need to meet the person who was

famous years ago that earned being a speaker now, who you need to be meeting are the other people in that room who will be maybe accomplished enough later to be the speakers years from now. You want to come up in the industry with those people, and that works better networking laterally than trying to hound the celebrity developers that everyone else is trying to fawn over. (Note that the tutorials events are often better for getting to know the other people in the room – at main conference talks that are more expensive people tend to scatter shortly after the talks and they rush off someplace else so we miss the chance to meet them.)

9

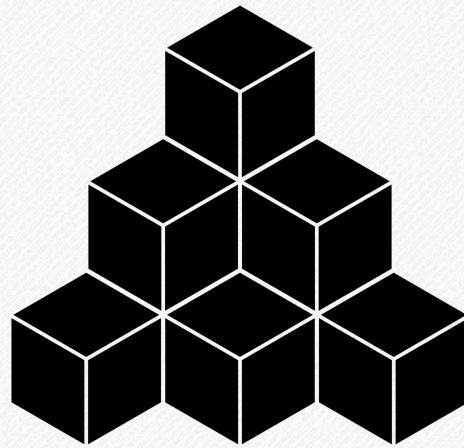


In the same way that we can sometimes study level design by investigating specific cases, one way into thinking concretely about gameplay design is to take apart games that we like, or worked on, in an effort to better understand how they do and don't meet their apparent goals.

GAME ANALYSIS

WHY MINECRAFT WORKED IN 2011

Minecraft took the world by storm. People who didn't care for videogames found that this game spoke to them. People that already liked games enjoyed it too! It's a perfect place to begin our dissection of games.



Background

If you already know Minecraft, skip ahead. If not, here is some information to give context to the rest of this section:

- The world is made entirely of 1-meter cubes.
- Any of those cubes, except bedrock at the very bottom, can be removed or moved, enabling the player to reshape the world.
- As far as the player can wander in any direction, more terrain will be randomly generated.

- Day and night cycles pass.
- Aggressive enemies spawn when and where it's dark.
- Underground there are cave systems with lava, iron ore, diamond, and a few other minerals.
- Above ground there are trees, hills, animals, streams, oceans, cliffs, sandy beaches, deserts, etc.

First-time play tends to go something like this:

1. The player spawns with no supplies.
2. Punching trees drops logs, which can be used to create a workbench, a weak wooden pickaxe, and a few other basic supplies.
3. The wooden pickaxe can be used to break stone, which can be used to make stronger stone tools. Stone pickaxes are then used for coal, iron ore, etc.
4. Coal and wood are combined to make torches, which cast light to prevent enemies from spawning in otherwise dark areas.
5. Combinations of materials produce a variety of other objects.
6. The player dies a few times, learns a few things, tries out a few different kinds of projects. Time passes.
7. The player then builds the Taj Mahal, or something very much like it.
8. Players start having Minecraft dreams.

My Narrow Experience

My first two weeks with Minecraft were entirely in single player; my next two were exclusively in multiplayer on a small server (about 12 regular players) with a cooperative, non-griefing community. For a day or two in between, I briefly explored huge servers.

By no means are my design observations below intended to be comprehensive. There are so many different ways to play, and in multiplayer, so many different communities to play with.

Here is what I managed to extract from my experiences with the game.



Modest interior decorating. Paintings are pre-made.

What Works

PRICE

Because online play in Minecraft takes place entirely on user-hosted unofficial servers, rather than massive centralized data centers as is now the norm for MMORPGs, there is no ongoing subscription cost. It requires only a one time payment to purchase the game, which is a fraction of how much typical games cost.

ONE-SIZE-FITS-ALL

Minecraft supports players with many different preferred styles. Whether you're into building houses in The Sims, fighting monsters outside in Zelda or Diablo, fleeing zombies within narrow corridors in Resident Evil, exploring vast landscapes in Oblivion, gaming alongside others in your MMORPG of choice (complete with the politics and leadership that entails), maintaining or trading crops in Farmville, chatting in the social community in Second Life, sharing videos with the creature creating community of Spore, or designing

things with basic electronics as a hobby, all that and more happens here. Some games let the player "play any way they want" by giving them 3 ways to approach a fight: stealth, guns blazing, or hacking and trickery. Here you can pretend you're in Lord of the Rings, then find yourself doing interior decorating or building calculators an hour later.

OPEN-ENDED

Minecraft allows a range of commitment. The lack of structured objectives means the player is free to travel light, coming and going in short increments to explore and adventure, or to set about gathering materials and laying out massive projects. This flexibility in possible session length makes it very easy to lose track of time – intending to hop in for 15 minutes to reap crops, then sticking around for hours from getting lost in a cave or carried away with a new project. How long it takes to play can grow or shrink according



By the end of my month I was taking on big projects.

to the player's interest and availability.

COMMUNITY OUTSIDE THE GAME

The game, at least in its current form, is famously harsh to newcomers. Night comes sooner than expected, enemies are more dangerous than they look, death is disorienting, and there's simply no information within the game about how to do basic, important things like making torches for light after sunset. Although traditional design suggests that this would be a huge flaw in the user experience, this increases the importance of being introduced to the game by friends, or of turning to the wiki and countless online YouTube tutorials. The user is not an island. Consequently, the player is either immediately pulled into the game's

tight knit player base, or becomes exposed to the tremendous wealth of information available outside the game.

EASILY SHARED

That Minecraft is a computer game makes it easy to Fraps a play experience for YouTube, or screenshot a completed building project to imgur. This sharing experience beyond the game gives players an audience, raising expectations. Meanwhile, those videos and screenshots function as viral marketing for the game, introducing outsiders to the game's aesthetics, concepts, and possibilities.



I made this based on the Library of Alexandria. On a server with many other players to share it with.

MOTIVATING PRESSURES

Building shelter that Creepers won't destroy, Spiders won't climb on top of, Zombies won't spawn within, and Skeletons won't fire arrows into gives utility to otherwise purely decorative elements. Doors, switches, fences, glass, armor, swords, arrows – along with landscape structures like moats, walls, sky bridges or defensive posts – become investments to protect a dwelling. The danger of darkness and quickness of nightfall give pressure and meaning to using daylight effectively. Without day and night cycles, and without enemies, the game would require much more initiative to learn. Instead, the game pushes the player to learn, without squeezing them through a narrow tutorial or precisely planned experience.

EARNED RESULTS

The difficulty and luck involved in finding materials, along with the risk of losing inventory upon dying and respawning gives value to the inventory. Crafting adds to this

sense of ownership: even though recipes are simple 3×3 grid arrangements, the feeling that you made the armor you're wearing, as opposed to finding it whole in the world, makes it yours.

FRAGILITY

Exploding enemies – Creepers – put structures at risk, which makes them particularly terrifying to a proud architect or red stone electrician. Fires can also spread, destroying flammable materials



This is my pickaxe. There are many like it, but this one is mine.

(lumber, doors, or nearby trees). Death by enemy, fall, or lava makes the player drop everything in inventory, which vanishes if not picked back up within about 5 minutes of respawn. This lack of

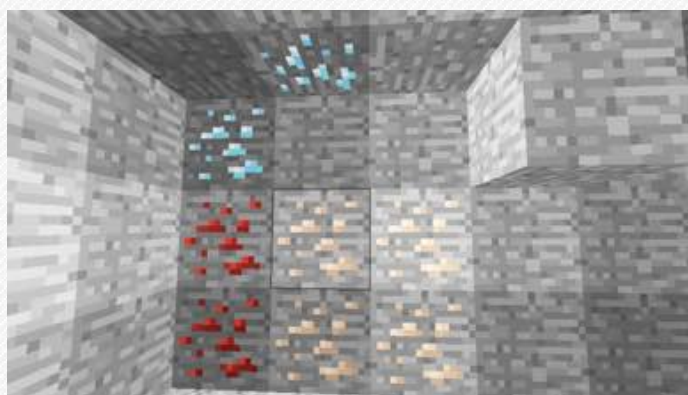
safety or permanence adds to the perceived worth of items and structures, since they have to be guarded by the player, and we have a tendency to grow fond of things that we're responsible to protect and maintain.

UNEXPECTED TENSIONS EXTEND PLAY

A side effect of fragility is the impulse to restore order after an accident, which unexpectedly extends gameplay time. We ideally want to leave the persistent world at a good stopping point, not worse off than when we started, so when inventory is lost upon dying from enemies, or once a wall gets blown apart by a Creeper, tension and dissatisfaction exist until the supplies are recovered or the damage is repaired.

RANDOMIZED REWARD (GAMBLING)

Variable ratio reward, the good ol' Skinner box mechanism common to many games in the form of randomized drops, dice rolls, and surprise chests, plays a central role in Minecraft addiction. Here, every single block removed



Top: diamond, left: red stone, middle: iron ore. Jackpot!

underground could reveal iron, gold, red stone, gems, flowing lava, or an expansive natural cave system behind it. There's excitement in the unknown, and in the sense that if just a few more cubes are chipped away, payoff for the effort will be found.

OVERSHOT/UNEVEN NEED

Game currency purchased through console downloadable games, card collection packs, or web social games are typically set up to have uneven leftovers after purchases. A lawyer even sued Microsoft over doing this for Xbox Live Points. This scenario comes about organically in Minecraft. For example, you have 26 logs, but need 35 to build the balcony you have in mind, so you set out on a

logging expedition – and while out in the woods with an axe anyhow, you bring home 25 logs. $26+25-35 = 16$, so after building the balcony, you now have 16 logs at your disposal, which probably isn't quite enough to do whatever you'll next want logs for. This mechanism gets amplified by rare, useful materials like red stone, or time-consuming materials like obsidian. In the past, I've discussed this mechanism as the "chips and salsa" design, so called because when the salsa bowl is empty but we still have chips, we refill salsa, and when the chips bowl is empty but we still have salsa, we refill chips, until one or the other runs out.

VALUE IN TRADING

All the reasons stated above adding to the value of materials – unpredictability in finding them, uneven supply needs, risk of their loss – combine to make them valuable for trade or gifting. Different biomes and varying distributions of minerals can lead



Not a very fancy house, but I bet he'd trade snow blocks for sand.

to complementary overstocking between players, with too much sand & glass stored by a desert dweller, snow cubes by another player in the tundra, and lumber for the forest resident. This means that the player who wishes to have a full palette of materials available for their construction projects is unlikely to be a total hermit, instead reaching out to exchange some local excess for what's abundant elsewhere.

ABSTRACTED VISUALS

Part of the appeal of pixel art, whether we're talking about Space Invaders or Passage, is that the simplicity saves our attention from being wasted on nuances like bump-mapping giving everything a wet plastic look. We don't see

cubes in Minecraft after only a few minutes playing: we instead see an ocean, a fortress, a tunnel, or a tree house. To someone new to Minecraft, screenshots of the terrain all look clunky and arbitrary. To a Minecraft player those same images look inspiring, magnificent, and ripe for settling.

LOW DETAIL = LESS ROOM FOR ERROR

The player's custom texture is 64×32 pixels. Whether players want avatars resembling their real-life appearance or their favorite

cartoon, comic, movie, or game character, it's hard to mess up too badly when drawing a 64×32 texture. That simplicity extends in an even more dramatic way into the world, where everything fits tightly on a 1-meter grid.

Alignment is easy, symmetry is easy, and the player is never paralyzed deciding between minor, fractional differences in dimensions. Anyone can throw together a decent little house in a single play session, and with practice and a bit of experimentation, come up with something both creative and presentable.

MASSIVELY MULTIPLAYER LEVEL EDITOR

From a purely functional perspective, Minecraft is a multiplayer real-time level editor. Combined with the simplicity of 1-meter block units, it's easy to create novel and engaging spaces for others to explore, between exploring the spaces created by others. It's MS Paint for 3D level creation.



INFINITE WORLDS

Players can – and do – throw away and initialize new random worlds until they're happy with where the central spawn location is situated. And no matter where the players start, walking far enough in any direction will yield major differences in the environment. Each random world generated has the potential to become almost twice the surface area of Earth, if players wander out in every direction.

UNCHARTED WORLDS (NO MAPS OR GUIDES)

A side effect of infinite worlds is that there's no single guide of quests and locations, which helps save the gameplay from becoming an exercise in following directions, as has become increasingly common among MMORPGs and single player games alike. For the same reason that the game thrives without a tutorial, many games suffer in playability from the prevalence of online tips sites. Though a game can be designed as though the internet does not exist, no players will be fooled;

Minecraft uses procedural map generation to counter the irresistible temptations that spoil our fun (we have no choice, really – because if we don't utilize online resources, other players will, leaving us behind). No one can give away the secrets of the world a Minecraft player is in, since every world's geographic secrets are unique.

LOW-FI MODS

Modding for feature expansion and new art is simple, with still more support for mods being built into the game's next iteration. Already people are adding more animals to the game, improvements to world generation, and inventing new modes. Though modding is possible for many commercial games, in this case Minecraft's low fidelity again becomes a benefit, since getting homemade mod content to blend seamlessly with the original game's visuals doesn't require a team of 3D artists and months of dedicated

effort. This enables mod creators to think and compete over design, rather than emphasizing production quality.

WATER AND LAVA SOURCES

When moving water or lava by bucket, it isn't the bucket full which is moved, but rather the source. Frequently, filling a bucket or two at the top of a waterfall can stop the waterfall – because the waterfall is inside the buckets. Likewise for lava – plucking up one source cube grants the power to drench a mountain or fill a narrow channel with lava. This design choice reflects a very unnatural contrivance which does wonders for gameplay, by removing the tediousness of moving liquids and instead allowing the player to take action at a conceptual level.

EXPLOSIVES

Without TNT, the game simply wouldn't be the same. Explosives are made of materials difficult to acquire – sand mixed with the sulfur dropped by the exploding

Creeper enemies – but they devastate caves, can be used to produce fearsome traps, and are an anarchy player's dream (well, that plus lava source buckets). Thanks to the respect for materials, inventory, and general property that derives from the time necessary to accumulate and put it all to good use, carrying even a few TNT items can feel a bit like how one might imagine it would be for a civilian to be in possession of a hand grenade. Gone is the triviality common to explosives in action games, and in its place is reverence for the weapon's potential to save or destroy time.

FARMING

The player can make a hoe, gather seeds, till grass, and plant wheat to be turned into bread. Reeds can be planted, then chopped down each time it grows to full height, for pressing into paper then books to join with lumber for shelves. Trees and cacti can also be planted, to farm lumber or



Wheat farm, for bread.



Reed farm, for paper.

more cacti. All plants require nearby water, light, and soil. Each of these grows at a different rate – which applies even while the player is away – and if they aren't harvested they stagnate at maximum height, causing the the player to miss out on potential materials from fresh growth. These motivate the player to compulsively stop in to harvest various crop types, and to make



A huge server was due to be reset on Dec. 20 for a major update. The night before, we were all given unlimited TNT and permission to destroy the world before its imminent deletion. The scene was surreal.

the rounds at the start and end of each session, further prolonging play.

DISTINCTIVE STYLE AND CONCEPT

Though both visuals and gameplay were cloned and advanced from the obscure game Infiniminer, when people see a screenshot, DeviantArt, or t-shirt with blocky people or cuboid worlds, that's now associated with Minecraft. Franchises like Halo, Max Payne, and Half-Life all have a distinctive lead character; every element in the Minecraft world – and even things not in that world but rendered the same way – is now associated with this particular game.

CROSS-PLATFORM SUPPORT

There aren't many commercial games built on a Java foundation. Other than Minecraft the only other prevalent example is... RuneScape? Yet the OpenGL support through the Lightweight Java Game Library enables unusual graphical complexity and performance for a cross-platform format. For a game that relies heavily upon word of mouth to achieve more sales, this avoids dead ends due to Windows, Mac, and Linux divides within peer groups. (This is likely more relevant among Minecraft's disproportionately geek audience than among mainstream computer users or game players.)

ACTUAL SANDBOX GAMEPLAY

Sandbox gameplay became a buzzword a few years ago, but bear with me. In the past, sandbox gameplay has meant anything from "the player can cause chaos between advancing the missions" (Grand Theft Auto), "the player has to cause chaos to advance the mission" (Just Cause

2), to "there are side quests the player can optionally complete, and in any order" (Fallout 3). In this case, the world is actually akin to a massive sandbox filled with huge grains of sand – millions upon millions of them – just waiting to be piled into castles, dragon sculptures, and rivers. Minecraft is a sandbox, in the most important sense of the word.

WISH FULFILLMENT

There was a time when the relative simplicity of homes meant they could be built entirely by the individual or family expecting to live inside. Sometime between log cabins and today's mountain of logistical paperwork, wiring, plumbing, etc. that largely came to a halt in the industrialized world. Minecraft gives players a way to design and build their own home. It's a similar enjoyment to architecting space in The Sims, and although the fidelity is lower, the player gets to live inside it, instead of merely observing other characters from the outside.

SOCIAL BONDING

Finding materials as a team, fighting enemies together, coordinating massive construction efforts, and seeing how the world transforms over time forms a bond with the other players. With each world being unique, and each server going through its own architectural history, a highly compressed feeling of growing up together is simulated. Players share common knowledge of location names, local happenings, and histories of peer personalities on the server. Thrown in with a bit

of hardship from Creepers, fires, and resource scarcity, plus current players assisting newcomers, a very dynamic and interconnected community of shared understanding takes shape.

SURVIVAL HORROR + SCARY AUDIO

The enemy sounds in Minecraft are not much fancier than the graphics. And yet, because of the danger they pose to the player inventory, the proximity of the enemies implied by the sound is genuinely terrifying. Player weapons are weak – arrows are scarce, swords are short range,



An underwater village built as a group effort.



*12 pumpkins, 14 moss stone, 8 gems,
19 brick blocks, 29 obsidian, 21 TNT, set of
diamond armor/gear... bad time to die.
The better you're doing, the scarier the game.
(Picture for sake of illustration; players stash
their best goods in chests before going out.)*

and rare materials are needed to make strong blades. Attacks are relentless, and come not only at night, but from any dark place, including a natural cave or a dark corner of the player's home. Since at any given time the player is likely carrying supplies either mid-gathering or mid-building, there's always something at stake when the player encounters enemies.

FORCED BREAKS FOR NIGHT

Short music tracks play when the sun is about to go up or go down,

signaling either the need to seek shelter or to prepare for adventuring out another day. During the sheltered down time at night, a break is imposed on play, giving outlet to web browsing, book reading, homework doing, TV watching, or even indoor Minecraft tasks (smelting ore, harvesting indoor farms, crafting, deep mining...). This helps keep the outdoor activity from feeling like it's completely dominating the player's time, and makes being outside more exciting since attention must be paid to either leaving enough time to return to shelter, or carrying enough supplies to create a provisional site on short notice.

CHARMING CUBIC ANIMALS

Boom Blox introduced the gaming world to cute blocky animals. Minecraft animals are in a similarly adorable, low-poly style. Finding cows hopping around a construction area, chickens wandering around a house, and sheep riding in mine carts is

delightful and unexpected, breathing life into an otherwise static world, without adding to the danger. That the animals yield useful materials when hit (feathers for arrows, meat for health, hides for armor, dyeable wool for colorful building and decorating) only increases the positive association with these animals, who remain unaggressive even while under attack.

KID-FRIENDLY

No swearing, no sultry cutscenes, no political or religious philosophababble, no blood, and no torture. A lot of games and media out there are almost kid friendly, but fail on account of carelessness on the part of the authors.

Minecraft doesn't need those gimmicks to work, so it doesn't have them, making it possible for families and young peer groups to adventure and build together.

SURPRISES IN UPDATES

If there's a roadmap of what features the developers intend to add next, it's kept secret. Each

update to the game includes undocumented feature additions – new materials, new crafting combinations, and other new features are thrown in unexpectedly. Halloween on year included a massive update, including a few minor touches like pumpkin blocks, but also a whole extra dimension, “Nether” (Hell) with new monsters and materials which can be accessed by making portals from the main world. More recently, music blocks were added, creating another opportunity for people to develop, teach, and show off talent within the game.

IN A WORD

Minecraft is a game about discovery. Discovering what's beyond the horizon, discovering new cave systems, discovering incredible projects others have done, discovering new features snuck into updates, discovering new like-minded people, discovering architecture, electronics, sculpture, texturing,

landscaping, action, photography,
decorating, music, trading,
storytelling, adventure, modding,
and discovering that we all love to
make things, provided that we
have an accessible and cost-
effective way to do so.

REFLECTIONS ON THREE SPECIFIC HOBBY GAME PROJECTS

Here are three team projects for which I served as a lead for code and core design. I developed these games with artists, writers, musicians, and other designers. Full credits are available in-game.



VISION BY PROXY: SECOND EDITION

Artsy Puzzle Platformer

Released August 2011

6.9 million plays by Feb 2014,
Best of SIEGE 2011 Student
Showcase

VbP:SE did better than we expected, making it to the front pages of several major Flash game portals in the US and abroad. We later developed a sequel (Ms Vision by Proxy) in which we attempt to address the game's brevity by involving more

character eyes. The short length of VbP:SE probably worked a bit to its advantage though, since it kept the scope such that players can get through the full game comfortably in one sitting.

Some people especially disliked the bonus levels, while other players specifically preferred those stages, both for the same reasons: they focus on platforming instead of puzzle solving. The game was short enough, all included, that players who like either were mostly able to slog through the



ones they felt so-so about. In any case it worked out well having these types of stages separated conceptually as a post-story second play through, rather than forcing both level types into the same timeline. It's a trick akin to what's seen in Portal or Mirror's Edge – at a much smaller scale, of course.

The game also would have suffered more from being too short if we hadn't milked the engine, level parts, eyes, and cinematic sequences to squeeze in twice the levels. Most importantly we were

able to get this increase in length without creating any new art assets nor needing to program any additional features. I just spent an extra day in the level editor churning out 9 more levels and throwing away the 6 I liked least to leave my preferred set of 3 remaining, which is a fairly common way that I like to do level design when time and tools allow for it.



MECHSTRIKE

Set-and-Play Sideview Action

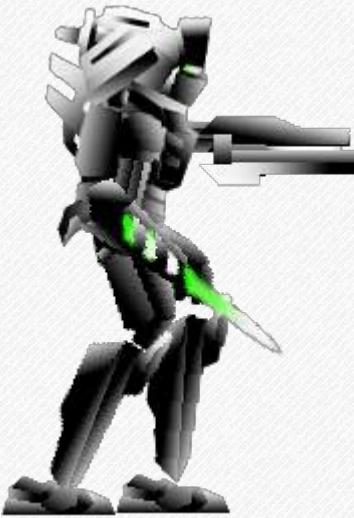
Released May 2011

127,000 plays as of Feb 2014

I feel like this game had just the right number of unit types: 11. More would have been harder to learn, and fewer would have limited depth and variety.

Early on during development we briefly explored what it would take to make a more complex and involved simulation with more detailed AI, stat, and strategy options, though we ultimately decided that something like Gratuitous Space Battles might

not make sense for an in-browser gaming audience. That worked out for the best anyhow, since we didn't have the time outside of classes during the semester to pull off something on that scale and polish. We made a lot of compromises to keep gameplay quick and to the point. In the end we made it simple and quick for players to get set up, which turned out to be vital for a game like this since trial, error, and observation is necessary to learn how each of the weapons and units work out against and alongside one another (from emergence, though, not from any contrived countering or boosting relationships).



It's surprisingly difficult to come up with cool, non-cheesy names for futuristic military units that haven't already been done to death.

Our writer came up with excellent names for each of the units, in addition to helping us come up with a varied set of weapons and abilities and writing the opening crawl. The techno-metal track we selected from Kevin MacLeod's Creative Commons library fit really well, but like any MacLeod music people hear the songs in YouTube videos and other games.

MechWarrior 2 was a source of inspiration for us, and the song we picked has a similar vibe to Dragon's Teeth from MechWarrior 2: Mercenaries.

Though my animation engine wasn't particularly great, our mech artist did a really nice job with the

individual mech body parts.

Without cool looking mechs, the game wouldn't have worked at all. Our main UI designer wound up doing all of our final tank and plane art near last minute (not her fault – originally someone else was going to make them), and thankfully they came out distinct and recognizable despite the time limitations. Our original plans included the ability for mechs to lose their limbs during battle, as they do in MechWarrior games, which we wound up cutting since our mechs didn't look as good without arms, and if mechs lost any of their few weapons during battle it would only drag out the endgame.

When there's only a few units left in a well-matched battle it can be kind of suspenseful... but it mostly just highlights the weakness and inaccuracy of my AI. In my defense, for about half of the project we had someone else onboard specifically to focus on AI programming, though on account

of outside obligations, he wound up parting before really getting started. That type of uncertainty is simply a risk of making hobby or extracurricular videogames, since we have no pay nor course credit to offer. The core gameplay still works with the mostly functional AI I stitched together.

We were considering having multiple rounds per configuration, with players able to make changes to their AI or weapon settings between rounds but not to which vehicles are included, but I'm glad that idea didn't make it. Part of the fun is quickly mixing up fresh scenarios, and being forced to watch a poorly chosen team lose again and again would be rough. The single round simplicity of our matches and how quickly they can be set up even made it possible for my friend Henry and I to play indirectly via Facebook.

The team had a bit of back and forth about whether the players should be able to do anything during the match. Proposals

included pressing a key to time deployment of a second wave of units, using the mouse to fire an orbital super weapon, toggling some AI variables, or directly controlling a lone commander unit. In the end we went with pure spectatorship, with shared camera control, which has a very different feel than being involved during the action by prolonging a sense of hope rather than demanding constant focus. Although we added little smoke and sparks to damaged units, we opted to leave precise vehicle health bars out of gameplay, partly for the uncertainty that their absence adds to excitement. Being able to identify the clear winner well before the end of the match, which could have messed up the experience, is much harder to do reliably since it's unclear how many more hits the units left on the field will be able to sustain.

Huge robots walking past one another mid-battlefield just looked wrong, and combined with their



slow speed we couldn't get away with it as we did with the tanks and jets. Instead, we programmed the mechs to stop and fight toe-to-toe with one another until one or the other exploded. All three mechs also have a special mech-vs-mech melee attack implemented: the humanoid walker slashes with the energy

sword (instant kill to any unit), the chicken walker kicks, and the spider performs a stomping action.

We didn't mean to make melee so rare as it came out, that happened only by late accident while trying to patch up other tuning imbalances. These attacks played a pretty pivotal role in the first build or two we released, with matches often coming down to mechs destroying each other point blank in the middle of the stage. The inevitable melee gave extra importance to fielding a humanoid walker. In later tunings (including the current version, 1.04) we improved the weapon accuracy for most units, decreased mech armor slightly, and increased the effectiveness of machine gun usage between mechs at medium range. These changes combined to make mech melee virtually never take place, though technically it's still there and will happen if matches unfold in a sufficiently peculiar way.

In hindsight, a way to save a launch team configuration for more rapid experimentation in sandbox would have been nice. Not only would it help players experiment faster, but it also would have made testing and iterating on game balance easier for us.

(continued on next page)



ZYLATOV SISTERS

80s-ish Arcade Action

Platformer, designed for shared screen co-op

Released Dec 2011

300,000 plays as of Feb 2014

Unlike most other games that I've worked on, after finishing work on this I still really enjoy playing it.

An experiment in modular game design to help reduce overhead,

Zylatov Sisters was successful in producing a good amount of variety and giving a number of beginning developers in our videogame development club a mix of experience.

Many players find it unacceptably difficult, though we understood from the outset that we wanted to make a game with pinball or classic-arcade pacing: constant vulnerability, typically 1-3 minute

sessions after a modest amount of practice, and replaying to earn increasingly higher scores.

Furthermore, we knew that the usual in-browser gaming audience might not necessarily go for this type of structure since it's an outdated taste. Players widely accustomed to videogames that are designed to be plowed through by persistence are presented here instead with levels that aren't meant to be consistently completed.

There's also a steep learning curve in that the arena-nature of each level means it's hardest right when each stage begins, when all enemies are still alive. Start the round carefully enough to survive and aggressively enough to cut down some of the horde, and with the remaining lives the odds of survival within that level improve considerably. Fewer enemies left in a stage also makes it easier to retrieve gun/item boxes, which can help improve survival odds for the start of the next level.

This was never meant to be a commercial game though, so we felt okay with simply making the game that we wanted to play.

Our decision to make any level equally eligible for selection as the first level gives the game variety for a high level of play and makes it easier for students on the team to show their work to peers. We also randomized which level shows up first, so that every player gets a slightly different first experience. However the lack of any intro, tutorial, or beginning levels served to further alienate players that found the game too difficult at first. We perhaps ought to have at least picked out a few of the easiest levels, and made the first level upon game start randomized from between those instead of all maps.

We also received frustrated feedback about the game's default controls, which we set with co-op in mind. Even though the controls can be reconfigured from the main menu, relatively few people seem

to be taking advantage of that. We perhaps should have gone with our original plan of using different default key mappings for single player vs co-op play, which we strayed away from late in development from fear that doing so would throw people off when trying co-op. Supporting that decision would also require a slightly different menu system, which we didn't have time left to do late in the schedule.

Unfortunately some players find the single player controls needlessly cramped, then simply quitting forever before either trying co-op or realizing that the controls can be easily reconfigured.

In other words: fully reconfigurable controls cannot make up for default controls that people don't like. Especially with a free game, people have no investment in making the experience work for them and can easily switch to something else, so if they don't immediately like the default mapping it's over – player lost.

On the plus side, more so than the other games I've mentioned here, people that do like this game seem to be returning to play it semi-regularly. That makes sense given the arcade model we were trying to emulate. By comparison though, it doesn't really make much sense to play VbP:SE more than once, and MechStrike's novelty and mechanics favor head-to-head play but don't offer much single player replay.

Our inclusion of co-op only support weapons worked out well for two-player, but may have strained our single player weapons offering a bit by comparison. We partly mitigated this by the three types of special weapons that can appear for getting a 5X or higher multiplier. However because it takes a bit of practice and luck to earn those, they don't do much to help first impressions. The two-player support weapons can be really powerful for assists if used effectively with the other player's help, though here again, tuning the

game to keep those co-op weapons from being overpowered rendered single-player even more unforgiving.

Co-op also lets players revive one another indefinitely. As long as at least one player remains alive and can find a chance to either finish the current level or kneel briefly by the fallen sister, it's possible to just keep going. Two experienced players working together can go on a roll for quite awhile, which in light of the game's harsh difficulty can be especially rewarding.

The aftermath stat and throwaway “award” screens (GoldenEye 007 style) worked well for giving players a moment to reflect on each round. Without them I think a hard jump from game over right back to level select menu would have been too jarring.

Outside testers really helped out a lot near the end of the project. I put out a call via Facebook and Twitter, and also had a few classmates try it out in person



where I could more easily observe and ask questions. It led to some important spot fixes to several stage layouts and assets, tweaks for more responsive controls (running then turning used to slip much more, as if on ice), and focused us on which wish list items were most important to implement with the time we had left (including saving high scores locally and the option to customize controls).

One of our members oversaw narrative design in addition to helping out as another technical game designer and artist. That can be a tricky role to play on a project that aims for an mid-80s action arcade aesthetic, but he

understood from the outset that the goal was low-fi retro, and he was flexible to the specific needs of the project. Since the genre is pretty light on written text outside of the intro (which players of this sort of game tend to skip anyhow), he instead improvised other ways to help establish the game's atmosphere: recommending the game's main font, drawing the title screen background to set the mood, and authoring his levels with a deliberate progression.

Our team members making chiptunes for this project really shined, pulling off a Castlevania, MegaMan-ish vibe pretty well in a number of tracks. I've been enjoying the soundtrack separately from the game. Tying specific songs to each stage helped give each level a consistent personality, beyond what feel we could achieve through changes in layout, tile art, and enemy, item, or spawn placement.

Of course, the best way to browse the game's music, see who's responsible for each track, and to find out more about the game is to play Zylatov Sisters (it's free - just search for it!).

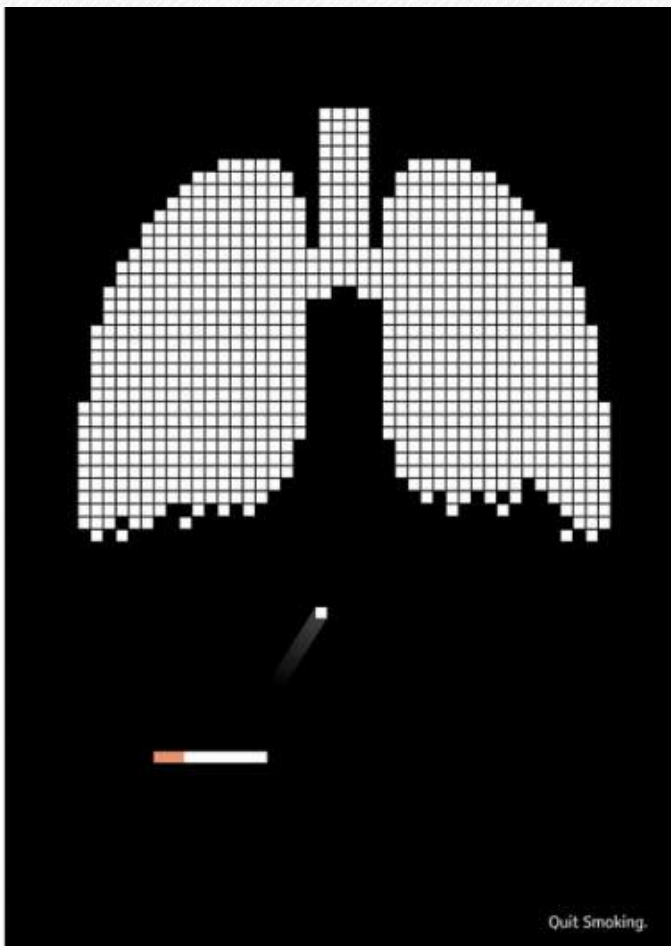
QUIT SMOKING: EFFECT OF INTERACTION ON INTERPRETATION

As a way to explore how gameplay can affect our 'reading' of a videogame's ideas, for this entry I built, shared, then reflected upon a playable adaptation of an image that began as a non-interactive work of art.



Not long ago, the image to the left, by the artist ReClark Gable, made its way around the internet. To be clear: this is not a screenshot. ReClark authored it as a still image.

This image communicates without interaction nor gameplay. Instead, visual parallels imply a mapping to components in the classic game Breakout, with lungs in the place of bricks, and a cigarette taking the place of a paddle. The idea of the picture is clear enough: smoking destroys the lungs, and



the reader/player is instructed to “Quit Smoking.” Through association with a classic videogame, it might even be read as suggesting that smoking is fun, or (drumroll please...) addictive.

This isn’t the first appearance of the concept, but something about this latest attempt resonated with people in a way that previous attempts didn’t, with the image receiving over 350,000 pageviews within a few weeks.

Piercey’s take, for comparison, received roughly 1/100 as many views per year as ReClark Gable’s received each week. Gable’s looks more like a playable game. The bricks fit tightly together. The ball’s trail suggests its movement. There seem to be enough bricks there to play for awhile.

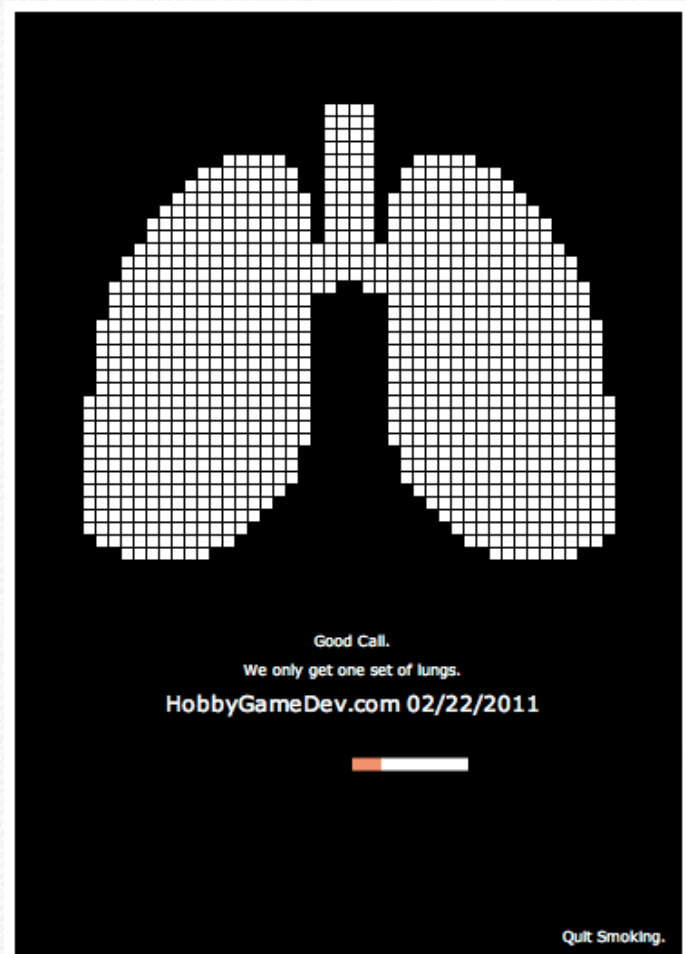
What if it did exist as a game? To answer this, I made a playable version.

To be clear: the goal of making this playable version was not to entertain, nor to persuade, but

instead to better understand the role of interaction in the communication and interpretation of meaning. We like to assume when we researching games with meaning that it’s significant that the artifacts studied are actual games, and not just static images.

IMPLIED GAMEPLAY CHANGES

There are a few changes to gameplay, which, though not derived directly from the image, seemed more consistent with the



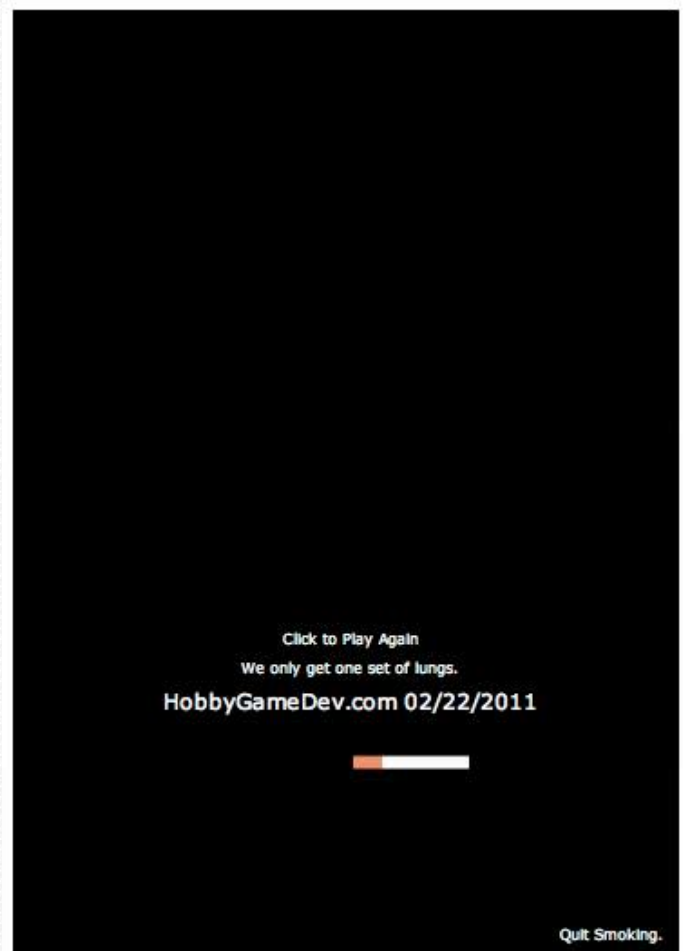
image's message than the standard Breakout conventions.

Without a lives display, an interpretation of one life or play (reset bricks each ball) and infinite lives or play (never reset bricks) would be equally consistent with the image. The latter design was selected, to represent permanent damage to the lungs. The former would seem to suggest that quitting smoking results in immediately undoing any damage done, as opposed to only ceasing the cause of further damage.

Traditionally, destroying all bricks a game like this one results in victory, advancement, or even just another set of bricks to be broken for points. However, an anti-smoking game (an unambiguous reading thanks to the “Quit Smoking” text in the corner) should not treat smoking until the lungs vanish as “winning at smoking.” Instead, the player can win by dodging the ball when the game starts – winning at smoking,

according to this implementation, means not starting.

Numerous other mechanics from Breakout were stripped because they seemed tangential to image or its message. Among them: paddle shrinking upon hitting the back wall, bricks worth variable points based on depth (scoring, in general), ball speed increasing due to depth reached or number of consecutive hits, and



discrete (rather than continuous) paddle angle segments.

To find out what people read into the image-as-a-game, rather than the game-as-an-image, I posted the game to Kongregate and Newgrounds.

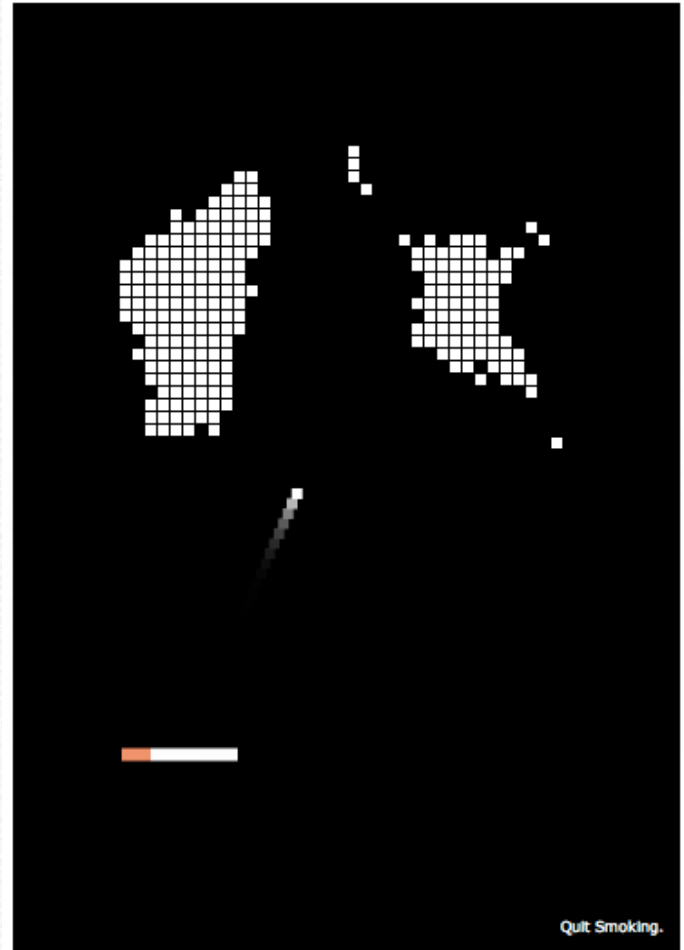
PLAYER ASSUMPTIONS

Many responses did not display an understanding of the nature of the work done here, suggesting that particle effects, power-ups, and so on would help make it a better game. This is of course partly a function of audience (self-identifying videogame players), and largely a matter of context (anyone at a Flash entertainment site is in the mood for entertainment, and likely to interpret their experiences through that lens).

This is somewhat unfortunate, because it means that many players, rather than interacting with the game in search of meaningful interpretation, set about to achieve the traditional goal of clearing all blocks, like so:

That takes nearly 40 minutes to do.

Not particularly evident in the original ReClark Gable image is



that drawing lungs at this fidelity requires a lot of bricks: 994. Compare that to the 104 bricks in Atari 2600 Super Breakout (8 rows of 13). Many of the comments noted the game's length, either as a criticism of the game ("too long"/"boring") or within the

context of the game's message ("Man it takes FOREVER to kill yourself by smoking"). Curiously, "FOREVER" here seems to be based on comparison to other online gameplay experiences, rather than real-time; if cigarettes made our lungs completely disintegrate after 40 minutes, that would be quite fast indeed.

Players that went for a cleared board also noted that, eventually, images took shape that look nothing like lungs. In the previous screenshot, I'm tearing away at what appears to be two island nations.

That point is correct: the longer the game is played, the more the concept seems to break down, or fall into the background. Does this mean that the opening set up – effectively, the still image – is doing the work, and playing only interferes with the message?

On the other hand, understanding the intended meaning of a game with a message often involves

winning at it. The longer someone plays this particular game, the more certain it is that they are not winning and have not caught on to the point. This was especially an issue in this situation since the feedback on player direction is delayed, subtle, and contrary to clear convention.

The original Breakout has an on-screen score display demonstrating that each brick removed represented progress, with bricks closer to the back wall awarding more points. In this way, the player gets coaxed toward hitting the back wall, which initiates a breakout in the classic game. (Achieving breakout by hitting the back in Breakout causes the ball to only bounce upward, instead of only downward, until it next touches the paddle. This mechanic was dropped for QuitSmoking as tangential to its message.)

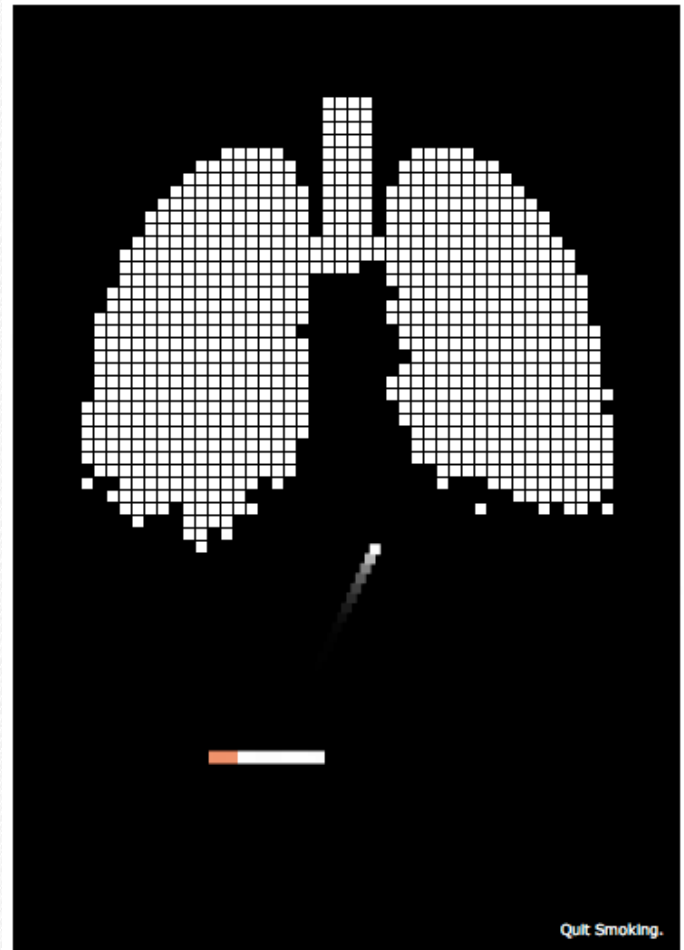
Because score tends to demonstrate incremental advancement toward some

greater reward or goal, including a rising score would not have made sense here anyway. However, an increasingly negative score, going down with each lung brick broken, could have spelled out more clearly and immediately to players that the goal of the game isn't to destroy all bricks. Without any score numbers to provide granular feedback, players acted on the assumption that what they previously learned through score reinforcement in similar games fits here too: brick removal means forward progress.

A SERIES OF GENERATED IMAGES

The longer the player keeps the ball in play, the more bricks get knocked away, producing a series of images, one of which (in theory) could be the exact same arrangement in ReClark Gable's original depiction. However even if that exact same brick configuration appears, is it experienced the same when we're the cause of it? If nothing else, there is surely a difference

between the image being one of many passing states, vs the totality of what is ever presented.



Also worth consideration is that the player has a purely subtractive effect on bricks. They can be taken away, but never added nor moved. A side effect of this is that only subsets of the starting bricks are possible images – combinations of the 994 bricks being either present or destroyed. Naturally, physical collision

mechanics make many of those theoretically potential configurations less likely than others (or impossible, such as a hole in the center of a thick ring). The lack of precise control over the ball tends to yield uneven edges.

Were a more geometric or clean figure hidden deliberately in the bricks, as say a key or a smiley face, even a few imperfections or incorrect hits could disturb the image. However since the hidden image in this game is the crispy, tar-destroyed lungs picture that we've seen in so many anti-smoking campaigns, any image of lungs with the bottom and edges unevenly carved away produces a similar enough image to recall the intended memory.

MISSING METAPHOR

As a quick aside: is the ball in this game "poisonous smoke"?

It is kept between the lungs and the cigarette, and it is what does damage to the lungs, so in terms of its role, the ball being smoke

functionally makes sense.

Representationally and behaviorally, however, the ball looks and acts in no way like smoke.

Does it matter whether it's smoke, or pure abstraction that steadily delivers damage to the lungs from the cigarette?

ATTACKING

The sensation of playing ball-and-paddle games is not so much like attacking, as it is like juggling. To this point, the original static image suggests a more direct attack than playing the game does. In the constructed snapshot it's on a one-way course for the lungs, though in play it's a constant back-and-forth.

Attacking isn't necessarily only shooting, Pinball games – digital as well as analog – have long contained enemies to be "shot" or "hit" with the ball. Because pinball paddles rapidly accelerate a ball, it seems more like an attacking mechanism than simply keeping the ball in play. Here, in Super

Pinball: Behind the Mask on SNES, the Wizard board contains an enemy that can be “shot” (top-right):



That digital pinball table is modeled closely after the layout of the real 1991 Terminator 2 pinball table, in which instead of a demon there's a T-1000 skull with glowing red eyes to be attacked with the ball.

Perhaps an anti-smoking game modeled after pinball, with cigarettes for paddles and lungs for bumpers, might hint at a more aggressive and direct attack on the organs than this Breakout metaphor.

CLOSING THOUGHTS

When Alleyway came out on Game Boy, and there were levels arranged to mimic the sprites of classic characters, did we think of the activity as killing Mario or Koopas? At some basic level, we of course realized that the part we were breaking corresponds to his leg, or his head, as when eating Animal Crackers, or biting into Ninja Turtles heads purchased

from the ice cream man. Though there is a dramatic differences between recognizing some part as represented, and thinking of the representation as the part represented.

Did any fan or player of Super Mario Land ever refuse to advance in Alleyway, not wanting to hurt Mario, or out of concern that doing so might cost a life?



IMAGE CREDITS

Cover and Chapter Icons

Cover Heart designed by Raji Purcell from The Noun Project

Ch.1 Power by Jardson A. from The Noun Project

Ch. 2 Teacher by Juan Pablo Bravo from The Noun Project

Ch. 3 Architect by Augusto Zamperlini from The Noun Project

Ch. 4 Tips by Lemon Liu from The Noun Project

Ch. 5 Gears by Edward Boatman from The Noun Project

Ch. 6 Mouse tools icon image in public domain

Ch. 7 Talking by Juan Pablo Bravo from The Noun Project

Ch. 8 Business Connection by Francisca Arévalo from The Noun Project

Ch. 9 Magnifying Glass by Yazmin Alanis from The Noun Project

Page Background Textures

Page texture from subtlepatterns.com

CC BY-SA 3.0 - Subtle Patterns © Atle Mo.

Chapter background pattern made by Olivier Pineda.

Getting Started

Making Your Own Videogames at Home is Totally Awesome - Working At Home by Rutmer Zijlstra from The Noun Project

How Long Does it Take to Learn Game Programming? - Calendar by Phil Goodwin from The Noun Project

Hobby Game Development: 20 Questions - Quiz by Rémy Médard from The Noun Project

Beginners Shouldn't Start with a Design Document - Document by Piotrek Chuchla from The Noun Project

Clone Videogames to Learn Real-Time Videogame Design - Space Invader by SuperAtic LABS from The Noun Project

General Concepts for Beginning Developers - Sketchbook by Edward Boatman from The Noun Project

Learn Videogame Development Like Woodworking - Tools by Dmitry Baranovskiy from The Noun Project

Fan Habits and Focus are Not Developer Habits and Focus - Crowd-Surfing by Hum from The Noun Project

Questions About Education

Advice to a New Student in Videogame Design - Education by Berkay Sargin from The Noun Project

Questions from an Elementary School Class - Students by Piotrek Chuchla from The Noun Project

Class Questions About Game Development Career - Image in public domain

Math for Videogame Making (Or: Will I Use Calculus?) - 3D by Michael Senkow from The Noun Project

Question About Comp Sci and Game Development - Network by Bruno Castro from The Noun Project

The Ways of Self Education - Education by Pete Fecteau from The Noun Project

Programming

Game Programming Fundamentals - Utility Knife by Hayden Kerrisk from The Noun Project

How Programmers Program - Code by Nikhil Dev from The Noun Project

Position and Speed Variables - Directions by Pedro Ramalho from The Noun Project

Float and Int Variables: Casting and Other Issues - Image in public domain

Hack then Refactor - Arrows by Juan Pablo Bravo from The Noun Project

Basic Real-Time Videogame Artificial Intelligence - Robot by Edward Boatman from The Noun Project

Quick and Dirty Backups - Image in public domain

Steps in Programming a Simple Real-Time Strategy Game - Image in public domain

Get Motivated

Stop Trying to Learn Everything Before Getting Started - Mind Blowing by Luis Prado from The Noun Project

“Overcomplicating Everything” - Confusion by Kelcey Benne from The Noun Project

Think by Building, Build to Answer Questions - User by Wilson Joseph from The Noun Project

Your Attitude Matters Even When Working Alone - User by Mundo from The Noun Project

Don't Wait for an Event, Job, Contest or Assignment - Waiting Room by Luis Prado from The Noun Project

The Brain is Not an Emulator - Thinking by Dirk Rowe from The Noun Project

Stop Arguing About What Makes a Better Game - Argument by Michael Thompson from The Noun Project

Start Before You Have an Idea - Image in public domain

A Little Planning Can Go a Long Way - Gantt Chart by Jeremy Boatman from The Noun Project

Game Design

Modest First Projects and Incremental Learning - Process by Joel McKinney from The Noun Project

Bottom-Up vs Top-Down Design - Create Database by Ilmur Aptukov from The Noun Project

Photographer's Algorithm - Camera by Okan Benn from The Noun Project

Doing More With Less: Short Videogame Design - Image in public domain

Colorful Oceans and Chunk Sauces - Noodles by Anna Wojtowicz from The Noun Project

Genres and Conventions: Known Patterns of What Works - Image in public domain

Level Creation

Anti-Design / Backwards Game Design in Goldeneye - Gun by Simon Child from The Noun Project

Level Design Concepts - Architect by Joel Burke from The Noun Project

Level Design Process - Mindmap by Leslie Tom from The Noun Project

Level Design Q & A - Image in public domain

Non-Essential Level Art is Essential - Graphic Design by Anna Weiss from The Noun Project

Visual Language in Super Mario Bros Level Endings - Flag by Ashley van Dyck from The Noun Project

Team Projects

Videogame Project Management for Hobbyists and Students - Meeting by Ainsley Wagoner from The Noun Project

Communication is a Game Development Skill, Part 1 - Image in public domain

Communication is a Game Development Skill, Part 2 - Image in public domain

Establishing a Videogame Development Club - Image in public domain

Industry

A Frank Look at Making Videogames Professionally - Image in public domain

Indie Game Development as Career - Art Collector by Piotrek Chuchla from The Noun Project

Influence of Business Models on Game Design - Pay Per Click by Siwat Vatatiyaporn from The Noun Project

What's Japan Doing Differently? - Japan by James Christopher from The Noun Project

Intellectual Property and Hobby Game Development - Copyright by Thomas Hirter from The Noun Project

Should I Release a Game as Soon as It's Done? - Checklist by Aaron Dodson from The Noun Project

How to Get the Most Out of Your First GDC - Conference by Wilson Joseph from The Noun Project

Game Analysis

Why Minecraft Worked in 2011 - Cubes by José Manuel de Laá from The Noun Project

Reflections on 3 Specific Hobby Game Projects - Thinking by Michael V. Suriano from The Noun Project

Quit Smoking: Effect of Interaction on Interpretation - No Smoking by Rahul Anand from The Noun Project