A Mini-Postmortem Roundup

PRINT

By Game Developer Magazine Staff

A reprint from the April 2013 issue of Gamasutra sister publication Game Developer magazine, this article rounds up several mini-postmortems for a variety of high-quality indie titles.

If there is one thing we've learned over the last year at Game Developer, it's that dev studios need to stay current on every potential game platform out there, or risk missing opportunities to reach the widest possible audience. That's why we've put together a collection of four shorter postmortems, each for a game developed for a different platform: Muteki's *Dragon Fantasy* (mobile), Subset Games's *Faster Than Light* (PC), KIXEYE's *War Commander* (social), and RSBLSB's *Dyad* (console). So whether you're a single-platform dev wondering if the grass really is greener, or you just want to learn more about what went right and wrong with a handful of standout games from last year, read on for the mini-mortems.

Mobile: *Dragon Fantasy*

By Adam Rippon and Bryan Sawler

We started on *Dragon Fantasy* on April 1, 2011 as a tribute to Adam's late father, Tom. Adam started making the game as a way to cope with the depression and stress in his life. While it probably wasn't particularly healthy to be as obsessed as he was with one project, he sure did get a lot of work done in a surprisingly short amount of time! The first chapter of *Dragon Fantasy* launched on iOS on August 23, 2011.

What Went Right

1. Regular Content Updates

The game was a modest success, and we immediately set to work on adding more content to it, hoping that by continually adding new content we could keep sales consistent.

While we weren't hugely financially successful from all of our free content updates, the goodwill and reputation that it earned us was a huge benefit. We've made a lot of friends in the indie developer community, which has been a huge help. We learned a lot about how to market our game via shows and via the press. Also, we bumped into Sony several times during the development of the game, and I believe that it was our dedication and cult-favorite status that led them to decide to include *Dragon Fantasy Book II* in the Pub Fund. Had we put out chapter one and called it a day, I wouldn't be writing this article right now!

2. Great Press Coverage

If there's one thing you absolutely need to have on your side, it's great reviews -- and we got lots of 'em. We enjoy a

4.5 star rating on both iOS and Android, despite the perpetually entitled rage of the "OMG WHY ISN'T IT FREE" crowd. We got great coverage from RPGamer, whose editor-in-chief absolutely loves the game. Joystiq gave us some great shout-outs. And our crowning achievement was our interview with Kotaku Australia -- Adam has a copy of it printed and hung up on his wall, and his mom even mailed a copy of it to his grandma. (It was that good.) Apparently it wasn't that common for Kotaku U.S. to run Kotaku Australia's articles, but they ran this one. Oh, and the sales bump from that beautiful article? Very, very nice. Great press goes a long way.

3. Good Tech Helps

Dragon Fantasy may not look like it's a super high-end engine, what with all the ginormous pixels and whatnot, but you'd be surprised! We've always rolled our own engine and tools, and the work on *Dragon Fantasy* was a serious boon to the production of our very powerful and very easy-to-use UI system.

While we didn't make a ton of money on the game itself, we did make a fair bit by using the tech we built for the game on other contract projects. We've done numerous paid projects for larger clients using our MuTech engine, even going so far as to use it in a political news app! And despite being reviewed by dozens of blogs, not a single one noticed that it wasn't a native iPhone app. We're pretty proud of that. So while we probably could have just done *Dragon Fantasy* with some off-the-shelf engine, there are some serious benefits to building your own cross-platform, application-agnostic engine if you have the means.

What Went Wrong

1. We Launched on Mobile

With each update we realized that it was getting harder and harder to reach more users interested in an 8-bit RPG on mobile. That's not to say they didn't exist, but it's very hard to inform gamers with more hardcore tastes about mobile games. That, and you can only rely on mobile game sites to cover your updates so many times. We might have potentially had more financial success by doing paid updates, but it would diminish quickly without a larger base.

Lesson one: Finding an audience for a $3 game on mobile is as hard as they say it is, even when your audience is more hardcore than the average mobile consumer.

And, speaking of launching: Our biggest problem, and one that still haunts us with *Dragon Fantasy 1* today, is the stigma of mobile games. Our game didn't feel like a crappy mobile RPG, but because a small team on mobile made it, we had a tremendous amount of difficulty getting anyone to pay attention to us outside of mobile. Our Steam submission(s) took months before they were rejected, and one of the other, more indie-friendly stores outright told us they weren't interested in mobile ports. (I was incredibly punchy that day, let me tell you.) Lesson two: If you're making a core game, launch on other platforms first to avoid being called "a mobile port."

2. Easy Porting Led to Undertesting

Since we loved working on *Dragon Fantasy* and really didn't want to see it end in obscurity on mobile, we started porting it to other platforms. The tech we had built up let us port the game quickly from iOS to both Mac and Windows. Unfortunately, the ease with which we ported the game led to us being overly confident about the state of those ports.

Being primarily a Mac shop meant that the Mac port was pretty heavily tested and has enjoyed a fairly stable existence. On the PC side, our testing simply wasn't sufficient and... Well, it just wasn't a great product at launch. The Windows game needed a lot of things that weren't necessary on other platforms -- things like an installer, runtimes, and checks to make sure appropriate versions of DirectX were installed. Add to that the different versions of Windows, ranging from XP ("just put things wherever, it's fine") up to Windows 7 ("sorry, you can't put those files there!") and we come to lesson three: Just because it works on everything else, doesn't mean you can cut corners on testing.

3. We Were Too Authentic

When we first started talking to the press about the game, we took pride in how authentic we kept everything. We stuck strictly to the range of colors the original NES was capable of putting out, and even limited our artwork to the number of colors-per-tile. The problem is, while as developers we appreciate that, and even a lot of the press we spoke to thought it was great, the masses didn't agree. We learned that what 8-bit games looked like, and what people *remember* 8-bit games looking like, are two very different things. Some of our maps could stand their ground against the very best-looking NES titles just fine, but still we saw cries about how bad the game looked.

Internally, we brushed these off as people who just didn't "get it" or weren't there in the mid-1980s to play the games we were inspired by. Our choosing to ignore this feedback just meant that we had turned more potential gamers away before we ever had a shot. Looking back over the feedback we received is part of the reason that when preparing our big "relaunch" of the game on PS3/PS Vita (and updates for existing platforms) we went back and updated all of the artwork to much better line up with what people want. There's nothing "not indie" about keeping your vision, but presenting it in a way to get the biggest possible audience!

Final Warrior Quest

If someone were to ask us three or more years ago what platform to develop their indie game for, it's almost guaranteed that I'd have suggested they target iOS. Now we have to face the irony that the platform largely lauded for giving small developers a chance with a cheap entry cost and removing the need for a publisher has so much competition that the best way to be really noticed is by getting a publisher behind your title who can devote a large amount of money to the launch. Finding an audience for your game on any platform is a challenge, but one as smothered by new releases as the iTunes App Store? Unless you're well known or incredibly lucky, don't expect to get a lot of traction.

And so, coming out of *Dragon Fantasy* we're not abandoning iOS as a platform altogether, but it's far removed from being the first platform we look to when planning our next games. The PC has made a huge comeback, and even the consoles are opening up a lot more to indies (we can speak firsthand about Sony, and from what we've heard Nintendo is a lot easier to work with as well these days). There are a lot of places to put your game. Pick one you can truly make it shine on and go for it.

PC: *Faster Than Light*

By Justin Ma and Matthew Davis

Faster Than Light (*FTL*) was a hobby project that tumbled into success. Here are our highs and lows from the development process -- and the Kickstarter campaign.

What Went Right

1. Unique, Novel Design

FTL started as a pretty basic concept. We felt that space games often focused on the act of piloting a ship or managing a fleet, but never let you feel like the captain. We wanted to experience making high-level decisions about the ship's strategy as well as managing every action of the crew. In our search for an enjoyable experience we created a unique type of gameplay: part simulation/strategy, part "Choose Your Own Adventure," and part RPG.

We found that for many people, the strength of the basic gameplay (with the help of amazing music from Ben Prunty) was enough to overlook what they might have considered flaws in the game: the simplistic graphics, its repetitive nature, and the sometimes frustratingly brutal difficulty. It effectively satisfied the little fantasy that we've all had when watching great science fiction movies and TV shows.

Initially we thought that there would be no potential market for a game this unforgiving, and we were genuinely surprised by how much other people liked it. In the end, we believe that the desire to create a game that we ourselves wanted to play allowed us to come up with something that appealed to others. Developers have stated this sentiment for ages, but we think it played an especially large role in *FTL*'s success.

2. Amazing Timing (and Luck)

In early 2011 we had both quit our jobs to spend a year making small game prototypes. After starting *FTL*, we used the IGF China 2011 submission deadline as a concrete milestone for our development; we decided that if we couldn't get a solid game prototype by that time, we'd move on to another idea.

We then spent four months working on game mechanics, rather than a game, and we were frustrated and unsure what the game would be until something clicked during the last two weeks. We determined the game's structure and pacing almost overnight, and were able to submit our first playable prototype to IGF, where it was well received. Indie game competition deadlines continued to line up nicely as periodic development milestones, but that was just the start of our fortuitous timing.

By early 2012, we were running out of funds and started to look into crowdfunding options. This was before Kickstarter had been used to raise millions of dollars for game projects, and at the time we thought it would be a good way to reach out to some people and raise a few thousand dollars. We had gained some publicity with honorable mentions in the 2012 IGF, which also gave us an opportunity to have a brief public demo on OnLive's service during GDC, so we were already planning to line up our Kickstarter with those events -- but our luckiest break actually came when Double Fine launched their Kickstarter campaign, which was two weeks before we were ready. Thanks to Double Fine, our Kickstarter got much more traffic.

So in just one week, the *FTL* Kickstarter benefited from the "Double Fine effect," two IGF honorable mentions, and a demo available on OnLive and the GDC show floor. This perfect storm of publicity undoubtedly led to our Kickstarter's overwhelming success.

3. Kickstarter: Positives

Money changes *everything*. The Kickstarter's success changed *FTL* from a hobby project to a business overnight. We became a "studio" with a decent number of fans keenly awaiting the release of our game. This was one of the biggest moments of success in *FTL*'s development, but also the primary source of stress and issues (explained later).

Our savings were all but used up one year into development. If we planned on releasing the game as a commercial product, we'd have to cover everything from food and rent to licenses and a lawyer, so we launched a Kickstarter campaign with the modest goal of $10,000, expecting that we could barely reach that amount, or, if very lucky, perhaps achieve $15K to $18K. But due to the fortunate timing of events described above, we ended up with just over $200,000 in funding.

The Kickstarter had a number of positive benefits on our development: First and foremost, we no longer had to worry about paying rent. We were also able to expand *FTL*'s scope; Ben was able to expand our initially slim music plan into a full soundtrack, and we were able to enlist writer Tom Jubert to expand *FTL*'s universe and lore, which in turn led to more ships, aliens, and weapons.

The campaign also publicized the game. Our campaign made us part of an expanding Kickstarter movement, and when we released, we became one of "the first" from that new wave of Kickstarter successes -- both of which led to a lot of press attention. It even got the attention of Valve, allowing us to both distribute the game and host our beta on Steam**.** This private beta was possibly the most important part of the Kickstarter; we got almost 3,000 beta testers, which is a lot for a new independent developer, and *FTL* became a far more polished and stable game for it.

What Went Wrong

1. Kickstarter: Negatives

With $200,000 in funding and nearly 10,000 new fans, public expectations are bound to change. Many people felt that with the extra financing, the game should be bigger and better than previously planned, but we had set a release date that was only five months from the end of the Kickstarter. Nearly every way to expand the project (hiring more help, licensing better technology, and so on) also required more time to get it done, so we had a hard time balancing the scope and time expectations. We feel that we were reasonably successful in that regard; the game was greatly expanded and released only two weeks later than expected, but it was far from a smooth experience.

The extra fans and publicity also meant a lot more public relations work, and since we didn't have a PR firm or marketing manager, that inbox full of email fell entirely on our shoulders. We set up a forum to start building a community space for our new (awesome) fans, which involved additional technical challenges we've never experienced. In retrospect, we should have had someone manage public relations (and our website) for us so that we could more efficiently devote our time to development.

In addition to contracting out help and building a community, we were setting up a company, finding a lawyer, discussing contracts with distributors, and much more. Every day we had to learn how to do things we've never had to do before. We were wholly unprepared for all of it, and relied heavily on the advice and help of our extremely generous friends and family. We set out to make a game and didn't realize we had to learn how to make a business, too.

2. Quantity of Events/Limited Development Time

Since *FTL* started out as a small experimental project, the original vision was quite limited. We thought we would have a dozen or so basic event types with a dozen different flavor texts for each type. Essentially, we wanted to create something similar to a deck of event cards in a typical board game. Our (perhaps naive) plan was to simply add more text if we needed more variety. How hard could that be?

After the Kickstarter, we decided to expand the game universe by adding a number of alien races, which meant more locations and events. At that point, we had something in the realm of 10,000 words worth of events. By the end of the project, *FTL* had nearly 20,000 words. That's *a lot* of text, but we discovered that even this amount would not be enough. When you divide the events between the sectors, even 20,000 words start to spread pretty thin.

One of the most common issues reviewers and players have with *FTL* is repetition of events. Even with a writer working for six months prior to release, we couldn't create the variety we wanted. While text is perhaps easier to create and integrate into a game than unique animations and art, it's hard to pump out the sheer volume needed to keep it fresh for hundreds of replays. Perhaps our time would have been better spent finding ways to make common events more compelling rather than adding tons of unique events that lose their impact after the first encounter.

3. Multi-OS Launch

Cross-platform development is generally a great plan; we wanted to support alternative operating systems and expand our user base to reach as many interested players as possible. But attempting a simultaneous, three-platform release for your very first game project is not a great plan.

We thought we were prepared -- *FTL* was planned from day one to be a cross-platform game, so all of the libraries and code would (mostly) easily transfer to other systems, and there was even a development version of *FTL* for Linux as early as five months into the 18 months of development. We believed, foolishly, that it would be an easy task to finish off the OSX and Linux builds once the Windows build was cleaned up and ready to go. But we underestimated the amount of work "finishing" *FTL* would entail.

The crunch of release involved the typical 12- to 15-hour workdays to just finish the Windows version, much less the last-minute cross-platform work. Even once we thought we'd succeeded, four days before the launch we discovered the Linux build wouldn't run on a substantial number of the different Linux flavors. Fixing the problems involved intense all-nighters and frantic phone calls to Linux-porting experts. We ended up launching smoothly, but it was hard-earned.

Supporting extra OSes causes problems during the immediate post-release support. We released two patches within a month of release to solve system-specific issues, and each one required making and testing four builds (Windows, OSX, and both 32-bit and 64-bit Linux) before we could release it. As new developers, our build pipeline wasn't ideal, and testing four builds with unique quirks with just two people doesn't speed up the process!

Simply delaying the Linux/OSX releases until two to four weeks after the Windows release would have made things more manageable. With a game under our belts and a build pipeline cleaned up, it wouldn't be as much of a hurdle in

the future, but stumbling into it clueless from day one was not ideal.

Social: *War Commander*

By David Scott

I've always been a huge fan of real-time strategy (RTS) games: It started with *Dune II,* continued into *Total Annihilation,* and peaked (for me) with *Warcraft III* and *Command & Conquer: Generals*. I played those games to death, and I met several good friends through those games -- including KIXEYE co-founder Paul Preece.

Inspired by the mods we played in *Warcraft*, in 2007 Paul and I had developed a few tower defense games in Flash that were successful enough for us to quit our jobs and make games full-time. Fast-forward three years, and we were working on a version of desktop TD for Facebook (*Desktop Defender*) and I was looking for some browser-based RTS games to play in my spare time.

The games I found were simple HTML affairs: You had slots you placed buildings in (the location of buildings didn't matter) and attacks were instant; all you saw was a battle report informing you of how many troops you lost and resources you looted. For someone whose favorite part of a RTS session was the epic battles toward the end of the game, this was a huge letdown.

We wanted to merge Tower Defense and RTS into one game; a game where the placement of your buildings and defenses mattered and, more importantly, where you got to watch and take part in the attack in real time.

In early 2010 I started work on *Backyard Monsters*, an MMORTS that built core RTS mechanics around a friendly and playful art direction (monsters building bases, instead of tanks and guns, in your backyard). As *Backyard Monsters* launched to rave reviews and our confidence (and player base) grew, we updated the graphics to be more realistic and edgy.

Emboldened by *Backyard Monsters'* success, I set to work on creating *War Commander.* It was to be an MMORTS game with nothing held back; we would have tanks, guns, airplanes, helicopters, suicide bombers, the works! In short, it was to be the game we were too afraid to make only 12 months earlier -- and since then, it has become the most popular MMORTS on Facebook, with the average player spending 8.6 hours in-game per week.

But when you go from "We should totally remake *Backyard Monsters* with tanks and guns," to a live game in six months with a four-person team, you end up cutting some corners with the intent of re-adding them after launch. We hadn't anticipated getting one million installs in our first three months, and we weren't ready for how quickly players would progress through the game, and so we had to spend the first few months after launch rapidly adding new content.

What Went Right

1. Keeping the Team Lean and Keen

We launched *War Commander* in six months with three people; that's pretty efficient! It took a year to get the headcount into double digits, and today we're still pretty lean compared to other game teams. We start all our games with just one or two people designing the game and making prototypes to prove out the mechanics, and slowly add tech and others to the team as the game gets closer to launch. Keeping the team small keeps communication overhead down to a minimum and, as a result, early progress is extremely fast.

Another thing we do right at KIXEYE is fully embrace the studio model. Each team is fully independent with the executive producer acting as the CEO of their own little company. What works for one team may not work for another, so we don't enforce certain methodologies cross-team. We do have shared learnings, but it's more organic, with people chatting over lunch to their counterparts in other teams, and weekly presentations to break down past releases and discuss upcoming features.

2. Making Tracks!

War Commander is two years old and still has weekly updates with new content and features. We now have five tracks in development: The first track is for large features that may take over a month to develop (the world map or customizable units, for example), the second track is for smaller features (anything that requires one or two weeks of dev time), and tracks 3, 4, and 5 are dedicated to improving infrastructure, fixing bugs and exploits, and improving operational efficiencies.

All five tracks work independently of each other, so if one is delayed it doesn't impact the progress on the others. We move developers between the tracks to keep things fresh for them -- it's good to have everyone on the team working on something that is player-facing now and then.

3. The Vertical Slice

With the exception of the world map (explained later in this article), we made the right calls on what to cut to get the game out on time and on quality. We had to sacrifice a few buildings to launch with air units, cull the number of infantry

units to add a wider variety of weapons, and cut music to have the units speak when you give them orders, but it was all worth it.

Making a vertical slice was key to this success. It forces you to develop each part of the game a little, instead of getting bogged down in one area. As an indie game developer, I know it's very easy to narrow your focus down to a very tiny and insignificant area of a game and just keep iterating on it. If there's only one thing you take away from this article, make it this: Build a vertical slice of your game ASAP, and play through it end-to-end before developing out any one aspect.

What Went Wrong

1. Skimping on the Metagame

One of the corners we cut to get the game out was the world map that you play in. In the early days of *War Commander,* you were presented with a list of targets (AI- and player-controlled bases). You could scout and attack anyone on the list, and killing them would remove them from your list and replace them with a higher-level target. The system worked well, but it was a poor substitute for a real world map, which is a game in itself. We didn't get around to adding a map until April 2012, eight months after launch. As soon as we released it we saw a 25 percent jump in engagement, retention, and monetization -- and realized how important the world map meta-game layer was (and how stupid we had been in not delivering it sooner).

2. Not Going Wide Enough, Soon Enough

For too long we went down the path of adding more buildings and units to the game to satisfy the needs of the late-game players. Instead we should have added features that create sandboxes they can play in. Instead of offering them new things to build, we should have given the players new ways to tweak existing content to make the game unique to each player and his or her army/base. This changes the game from "I have everything built and at max level" to "I have everything at max level, but maybe this isn't the best combination of things; I need to experiment!" This attitude has the benefit of creating a wider variety in base designs and attack strategies, making it more varied for the attackers and defenders.

3. Poor Time Estimates

We had a solid RTS game; you had your base, your army, and a world to dominate, but there was one bit missing: real-time battles. For many technical reasons we didn't allow attacking between two players that were online at the same time. We fixed this in October 2012 with the launch of Live Battles, so now players who were online at the same time could take part in the same battle and interact with them in real time. We knew this was not going to be something that would drive monetization in a big way, but we did it anyway because it was cool and we were not happy putting our name to an RTS that didn't have synchronous PvP battles.

We knew it would be hard to retrofit synchronous multiplayer into a live game while continuing to develop and release new features, but we had no idea how long it would actually take. We had to rewrite a very large portion of the game in a completely different language (a language none of the existing dev team was familiar with) so it could run on both the client and server. This took us a painful, drawn-out six months or so to develop, test, and release. If we had done it before we launched, we might have been able to save ourselves a few months of work, and if we had taken the time to sit down and correctly break out all the tasks and estimate them, instead of just diving into production, we could have better planned our time. The one silver lining to come out of it all was a new process on the team that has helpec us hit every deadline since the launch of Live Battles with a pretty high level of accuracy and minimal crunch.

Console: *Dyad*

By Shawn McGrath

Developing *Dyad* was a long and difficult process.

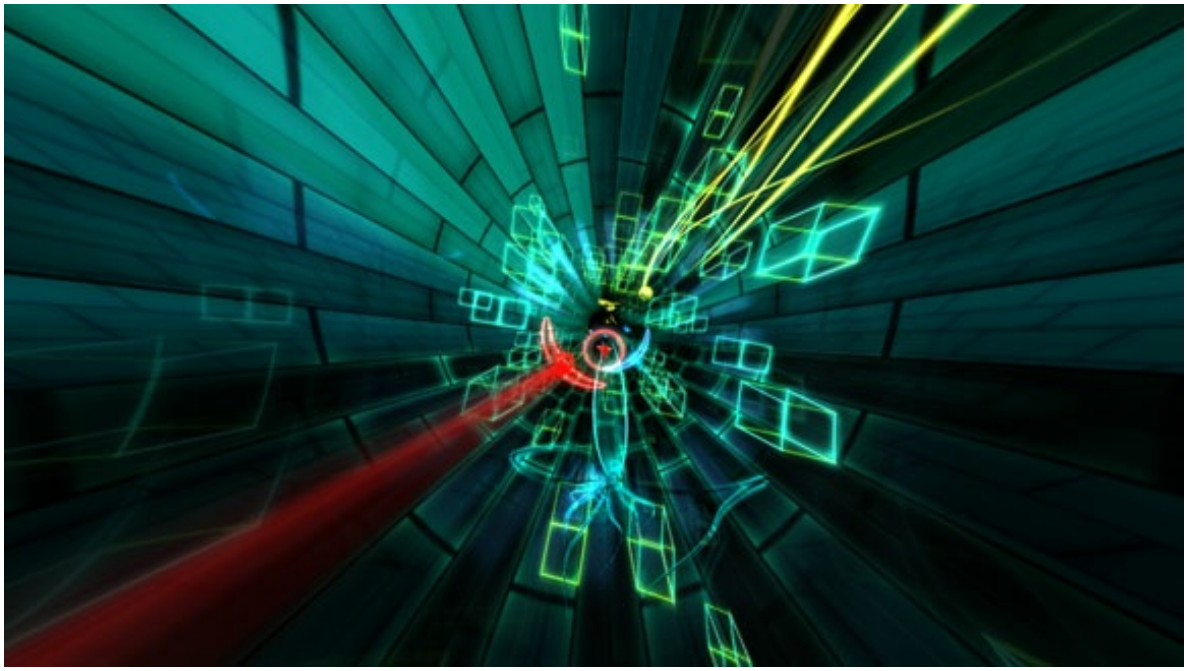
It all started when I was playing a lot of Kenta Cho games; I liked *rRootage* and *Parsec 47,* but *Torus Trooper* wasn't doing it for me. So Pekko Koskinen and I dithat ssected *Torus Trooper,* along with some other racing games, discovered a few design issues most racing games had in common, and decided to make a game that fixed those issues. We wanted to make a game that encouraged you to think tactically, instead of forcing you to rely solely on reflexes and muscle memory as the game got faster, but we didn't want to complicate the controls unnecessarily.

We figured we'd solve these problems and make *Dyad* in a year. We were wrong. Pekko eventually left the project, and I continued on for over three years trying to solve seemingly unsolvable problems. I worked 10-16 hours a day, six or seven days a week, for three years straight. It was worth it.

What Went Right

1. I'm Too Picky

Before making *Dyad,* I heard stories of Miyamoto's ability to discard months of work if it wasn't working out, and thought it'd be easy to do the same. It isn't. I threw away 90-95 percent of all the work I did on *Dyad*. I spent nine months on a mechanic that followed and replaced zip lines called "gates." Gates contained more variation and interesting gameplay than the rest of the game combined, but the mechanic was completely unteachable. I couldn't even teach my wife how they worked after several hours of one-on-one tutoring (and she's very good at *Dyad*), so I scrapped them. I created over 200 other levels, and I threw all of it away except for the very best stuff. As it is, the game is longer than I'd like, but I can't cut any more of it.



2. Doing (Almost) Everything Myself

In 2010 I showed the game to several publishers, and three showed serious interest. We negotiated getting funding and staffing up. I called it all off and decided to pay for it myself by living as cheaply as possible and draining my life savings. That way, I could make the game I wanted with as much time as I needed.

I did all the programming, game design, and graphics essentially by myself, except for some help with the PS3

version and a part-time co-designer for the first year of development. By programming my own engine, I was able to get the load times down to <1 second, and I was able to use low-level graphic effects and experiment with a wide variety of new graphics techniques. I used one monitor for Photoshop, one for code, and one for the game, which let me change the graphics and the code and see the changes instantly. Without this, I wouldn't have been able to come up with the visuals in *Dyad*, and without the live code update, I wouldn't have been able to test out nearly as many game design ideas.

I also did all my own promotion and advertising. This started when I built a large motorized chair ("THE MACHINE") that tilts and rotates to match the player's in-game actions, and went to PAX East in 2011 with the disassembled chair packed into my Chevy Impala. Jason DeGroot and I spent two days assembling it before I showed *Dyad* for the first time to a large audience with THE MACHINE. I received a lot of positive press from PAX East, so I decided at that point to do all the marketing myself. Even though it was very time-consuming I wouldn't have done it any other way.

3. Music collaboration with David Kanaga

David Kanaga really pushed what's possible with *Dyad*'s reactive music system. We worked for four months on the first track: He was in Oakland, I was in Toronto. He'd send me stems of the song he made for the level, complete with chord changes and interaction event sounds, and he'd explain to me how he wanted it mixed and how different game elements should change the mix. I'd send him a video of me playing the game with his system. We did this back and forth for four months until he was able to fly out to Toronto where we worked on the first 15 levels or so. He went back to Oakland, this time with a computer capable of playing the game, and sketched out the final 12 levels, then came back to Toronto to finish everything.

Most people don't believe me when I tell them the music was added last in the game. The game was essentially done with no music, and David was able to create unique music rules for each level to match the game rules, and write 27 unique songs!

What Went Wrong

1. Speed Makes Things Complicated

Dyad is one of the fastest, most complicated games I know; players must track the location and state of their avatar, their immune status/combo status/polarity, and the type/state/location of each enemy. It's almost impossible to process it all without a million tiny elements designed to help you, and if one of these elements isn't perfect the game completely breaks. I'll explain a few of those elements.

Representing depth on a 2D screen is hard, especially when moving quickly. There are two primary elements in the tunnel that make it easier to discern depth: Enemies will leave a faint highlight on the tube, and most enemies also have a trail drawn in front of them.

The player's "space squid" avatar is visually designed to be entirely functional. The bright main color and black/white circle in the center is a purposefully vague representation of its hitbox. Its physical size constantly shrinks and grows. When grazing, the player is about 10 times smaller relative to the enemy in the center of the graze circle than they are relative to normal enemies. When lancing, the player is about 100 times bigger. The player can also be two different sizes depending on polarity.

All of this information is hidden; the player's center is just a vague indicator of position. The trails behind the player exaggerate lateral motion in order to make it easier to see where the player is and where the player is going. *Dyad* is far too fast for the player to even look at their avatar, so I designed the trails to make it easy to perceive the player's position and motion from peripheral vision only.

The tube design was another important area in maximizing information processing. The tube acts as a reference point

for all objects in the game, and needs to feel like a fluid space while looking pretty. Most levels use a grid pattern to make it easy to discern distance and enemy patterns, and to see what players have lined up.

In *Dyad,* most levels have a double tube to enhance the perception of lateral speed in order to match the hyper-exaggerated depth speed -- if you focus carefully on the player vs. enemy speed, you'll notice the game isn't nearly as fast as it feels. The outer tube is offset such that the inner and outer tubes line up at the bottom where the player is. This increases the visual noise, and draws the eye to the area directly in front of the player. Blending is used to "white out" the area in front of the player to make it easy to see what's going on.

There are many more visual design techniques in play to make information processing as efficient as possible. I didn't expect there to be so many restrictions and would have loved more freedom in the visual design space.

2. Hard-to-Teach Mechanics

It took me a while to realize that *Dyad*'s mechanics are fucking weird. I showed the game for the first time at a Scott Pilgrim launch event expecting everyone to be able to pick up and play it; they couldn't. Then I added a reasonable tutorial, and showed it again at an Ontario College of Art and Design event. Still unplayable. It took over a year of near-constant playtesting before anyone could play it without assistance, and at least another year before it had a reasonable learning curve.

The entire game is a tutorial. I had to split the mechanics up into small pieces and teach them individually. The mechanics are boring in isolation, so I had to come up with a bunch of unique goals and modes to keep the game fresh and interesting, with varying subsets of the mechanics available to the player at any time. In the end, this made the game a million times better than it would have been otherwise, but it was extremely irritating to see people completely unable to grasp what I thought were simple concepts. The inexplicable nature of the game led me to a nervous breakdown in early 2011 before I revamped the entire structure of the game.

3. Communicating What *Dyad* Is

I really can't describe what *Dyad* is, which hurt sales more than anything else. I made *Dyad* to be as "pure" of a game experience as possible, without relying on tropes from other mediums. Communicating the things that Dyad does to your brain while playing is impossible without playing it. I got a lot of inspiration from Vertov's film Man With a Movie Camera, which as a movie meant to do only things that are unique to film. I think I did that with *Dyad*, which made it hard to talk about (and therefore very uninteresting to most people). I wish I could have come up with a way to talk about the game without compromising its game-ness.

Return to the full version of this article