Postmortem: Intelligence Engine Design Systems' City Conquest

PRINT

By Paul Tozour

Intelligence Engine Design Systems and our first game, *City Conquest*, came about as a result of a surprising personal journey.

My career began at Brøderbund in 1994, working as the lead designer on a real-time strategy title. After it shipped, I conceived a design concept for a new game -- a huge, epic, insanely ambitious dream game that I am still working toward even to this day. I knew that it would be possible to make it a reality someday, but I also recognized that I wasn't yet qualified to design it. My dream game required a huge amount of artificial intelligence, and I needed a much deeper understanding of AI to have any chance to design it properly.

I spent well over a decade learning everything I could about game AI, helping to develop the AI in games like *Metroid Prime 2* and *3*, *MechWarrior 4*, and some others, writing articles on game AI, evangelizing the use of navigation meshes for pathfinding, speaking at AIIDE conferences, and generally pretending to be a programmer while picking up everything I needed to know to design that game. Along the way, I got a golden opportunity to learn from a few of the industry's most acclaimed designers.

At the end of that process, I finally did learn what I needed to know to design the dream game. Intelligence Engine Design Systems is still working toward that game and growing toward the funding level and company size that can make its development possible.

But there was an unexpected revelation along the way. I had not expected that AI would change everything I thought I knew about game design.

The more I learned about AI and game design, the more I began to understand that they are two sides of the same coin. Game design is the creation of possibility spaces in which players can act; game AI is about developing systems to explore those possibility spaces and act within them -- usually on the part of game characters or entities, but sometimes by separate tools. When used wisely, AI can give us an incredible number of tools and techniques for improving our designs and helping us explore the ramifications of our design decisions more quickly.

The more I looked at the kinds of problems we encounter in game design, the more I realized just how many of them are fundamentally either decision optimization problems that are eerily similar to the problems we solve in artificial intelligence, or combinatorial optimization problems that we can optimize with the kinds of decision modeling processes already used in many other engineering fields. As a result, there are some fairly remarkable unexplored possibilities for different kinds of tools we can create to optimize the design process.

Game design is a process of learning and exploration. All game designs evolve, and even the most focused and well-planned projects spend a considerable amount of design iteration to get the design just right. Well-run projects are able to manage their exploration of the design space in a way that balances design evolution and its impact on product quality against all of the costs of that design iteration and experimentation. Less well-run projects get lost in

creative thrashing, unfocused design exploration, or perpetual preproduction.

When we design, we are struggling in the dark. We have endless options and ideas for design changes that can make our games more fun, engaging, immersive, or addictive. But we have very little ability to accurately predict all of the countless ramifications of these changes and additions to a game's design, or to truly understand how the game's character will change without actually implementing the changes we're considering.

We have precious few tools to explore that space beyond our own imaginations and our ability to actually go ahead and experiment, prototype, and test our ideas.

Professional aircraft designers have powerful engineering tools at their disposal to simulate "virtual wind tunnels" that can estimate the performance characteristics of their aircraft. Long before they ever need to actually build a prototype and place it in a physical wind tunnel, aircraft designers can quickly and cheaply model an aircraft using CAD-like tools and instantly see how its design influences its performance characteristics.

There is no equivalent for game design. We have no tools to show us all of the ramifications of our core design decisions. I believe we need to grow beyond the current, purely anthropocentric approach to design and accept the need for a process that involves some level cooperation with machine intelligence as so many other industries have done. Our industry doesn't need a "Photoshop of AI": it needs a virtual wind tunnel for game design.

So I decided to make one.

City Conquest began in August 2011, while I was earning an MSE degree from a University of Pennsylvania technology management program co-sponsored by the Wharton School. As the result of a market analysis in a Wharton marketing class, I identified a market opportunity for games that combined elements of tower defense games like *Kingdom Rush* with the depth of real-time strategy games like *StarCraft*.

I designed *City Conquest* as a hybrid TD/RTS to combine the simplicity, accessibility, addictiveness, and feel of tower

defense with some of the depth and strategic elements of an RTS. It needed to feel like a head-to-head TD game, with each player given a Capitol building to defend and special "dropship pads" that produce new units every turn.

The goal was to limit the scope to a one-year development cycle while ensuring that it was a serious game development effort with genuine potential to grow into a franchise, rather than a quickie game for the sake of "learning." In today's saturated mobile market, there's no point wasting time on any development effort that isn't at least making an honest attempt to be ambitious, polished, and unique.

Our goal with *City Conquest* was to optimize the "cost-to-quality ratio": the best possible product at the lowest possible cost. Our years at Retro Studios / Nintendo imbued us with an obsessive focus on product quality and polish and a conviction that this diligence pays serious long-term dividends.

At the same time, we were determined to reduce our risks by minimizing our overhead -- we would avoid growing the team any more than absolutely necessary, and we would outsource all of our art and audio needs while handling the design and production and the majority of the engineering ourselves. In large part because of this risk-centric attitude, IEDS maintained a very small team throughout development of *City Conquest*.

We had a basic playable prototype running within a month. The concept worked better than we had any right to expect. Our decision to use a fundamental gameplay to drive the game's development, rather than any particular narrative arc or artistic "high concept," went a very long way toward ensuring that the game would be fun and playable.

Production has gone amazingly smoothly compared to even the most well-run triple-A projects we've worked on. There were a few missteps along the way, and some clear failures of due diligence, but nothing that negatively impacted the final quality of the product. The fun began to shine through in the first few months of development, and I increasingly found that I was so genuinely addicted to the game that playing it became a dangerous distraction from actually working on it.

We took it as an encouraging sign, and this more than anything pushed us to see *City Conquest* through all the way to the end.

I have worked with some teams that spent months to years trying to "find the fun" during development. This is confusing. If you haven't found it yet, why are you in development at all? How did the project get greenlit in the first place? Why are you even calling it development when you're really still in preproduction?

City Conquest was completed several months later than expected, but most of this delay was for the right reason: because we were determined to optimize for quality and had the time to make the game better. Although we were careful to limit the scope of the project by adding new features only when truly necessary, we spent a great deal of time and effort on polishing existing features and addressing our playtesters' feedback. We were consistently unwilling to pass up changes that would improve quality, schedule be damned.

What Went Right

1. AI-Assisted Design Process

Our AI-based approach to design paid off in spades. There is no question in our minds that this exceeded our expectations and improved product quality and time-to-market. This was not some high-minded academic fluff but a practical, boots-on-the ground competitive advantage.

The full theory behind our design process is much too complex do it justice in the scope of a postmortem article. We hope to be able to explain it more fully in the future if time permits. But we can touch on two main elements of our design approach that impacted *City Conquest*.

The first was our optimization-based approach to the core design. The defensive towers and unit types in *City Conquest* were not based on arbitrary creative decisions: nearly all of the decisions around their functional aspects were guided by an explicit decision modeling process.

We clearly specified the design goals and constraints for all of our towers and units and then constructed a decision model to optimize the features of all the units and defensive towers together as a whole.

We then used an evolutionary optimizer to select the best set of nine towers and nine units that would best work together in concert to satisfy our design goals while remaining within the boundaries of our design constraints.

This approach is broadly similar to the one described in the book Decision-Based Design, although our approach is much simpler and customized for game design decisions.

We firmly believe that this type of optimization-based approach can pay major dividends for game developers. It can help us provide more value to our customers by reducing the complexity of design problems, allowing us to make more optimal decisions more quickly, and in some cases, allowing us to solve problems that are otherwise unsolvable.

The second advantage was Evolver, which we discussed in an earlier interview with AIGameDev.com. Evolver was an automated balancing tool based on coevolutionary genetic algorithms. Every night, it would run a huge number of simulated games between red and blue opponents, with each opponent evolving a "population" of scripts (each script being essentially a fixed build order of buildings within the game).

Evolver would generate random scripts, play them against each other, and assign them a fitness score based on which player won and by how much. It then applied standard evolutionary operators such as crossover and mutation to genetically optimize each population.

This meant that we could wake up every morning, open Evolver, determine which scripts the red and blue players ranked the most highly, and then plug those into the game and watch them play against each other. This instantly told us how our game balancing was working. Were players building too many Rocket Launchers? Not enough Skyscrapers? Were Commandos not useful enough, or were they consistently preferring Crusaders over Gunships?

We could then use this output to tune and refine a few units and buildings every day, tweaking their resource costs, health, speed, damage, rate of fire, and other parameters. It was very much like having an external outsourced testing team that would play the game overnight -- except that it was cheaper and more scalable than a human playtesting team, in addition to being capable of absolute objectivity.

We optimized Evolver by disabling the rendering, adding task-based parallelism, and hand-optimizing the gameplay logic. This allowed us to achieve roughly 1 million simulated games per 12 hours. We later upgraded the genetic algorithm to use an island model and hand-tuned the fitness function to achieve certain outcomes (such as helping the script populations learn to upgrade their Skyscrapers quickly enough to achieve optimal long-term income levels).

This might seem like a lot of work. It wasn't: the work to create, tune, and optimize Evolver was roughly two weeks' worth of development time in total. Considering all the valuable feedback that Evolver gave us, the fact that it gave us better results than we could have achieved by hand, and the fact that doing this initial hand-tuning would have taken far longer than the two weeks we spent on Evolver, we consider this an obvious net win.

It also left us with a system that we could quickly run to test any possible changes to the gameplay parameters to see the ramifications of changing any design changes -- in one case, we were able to quickly identify the issues that would arise from reducing the Roller's cost from 2 crystals to 1 crystal (and reducing the Roller unit's stats accordingly), and Evolver allowed us to immediately identify the problems and abandon this idea before it caused gameplay problems.

We also benefited enormously from having a large number of playtesters in our TestFlight team for eight months leading up to release giving us the invaluable aesthetic, usability, and other subjective feedback that Evolver could not. We eventually invited all of our Kickstarter backers to join us as playtesters.

As a result of all of these factors, the game was fun almost from day one. Every design concept worked.

2. The FBI: Fix Bugs Immediately

I've worked on several projects with 5K+ bug counts in the past. Once the team moves on to the final "bug fixing" phase, there are inevitably dozens of awful discoveries: "Wow, if we'd known that bug was there, we could have saved so much time!" "That bug caused ten other bugs!" "If we'd gotten that bug fixed six months ago, we would have totally changed the way we designed our levels!"

On one memorable project, a producer told his team: "Don't fix bugs -- we don't have time right now! Save them all up until the end!" The project failed catastrophically months later, destroying the studio and nearly pulling the entire franchise down along with it, in no small part due to the overwhelming bug count at launch.

That should never happen. Letting any software defects linger is a risk to the schedule and to the integrity of the code base.

Our approach is to fix bugs immediately. We don't work on any new features or continue any other development until all known bugs are fixed. Design flaws and performance problems count as "bugs." Playtesters' suggestions also usually count as "bugs," especially if more than one playtester reports the same problem or suggests the same change.

Our running bug count remained under 10 at all times throughout development.

Now that we've done it, we can't imagine developing any other way. What's the point of postponing fixes just for the

sake of the "schedule"? It's essentially a form of accounting fraud: you're pushing off a problem onto one side of the ledger to pretend that the other side of the ledger is doing fine. You're introducing hidden costs by sacrificing product integrity.

Our playtesters frequently mentioned the relative lack of bugs, and this codebase integrity ensured our testers would be focused on gameplay rather than technology issues.

In our experience, the practice of pushing bugs into a database and waiting to fix them until a later date is a major cause of product delays, developer stress, work-life imbalance, and studio failures. It's a primitive and barbaric practice and it needs to end.

---

3. Discovery-Driven Planning

While I was earning my Penn/Wharton MSE degree, I had the good fortune to attend a few lectures by Professor Ian MacMillan, coauthor of Unlocking Opportunities for Growth, and learn about discovery-driven planning. The lectures were a huge eye-opener.

The discovery-driven planning (DDP) process is explicitly tailored toward innovation-oriented projects and focuses on reducing uncertainty. For all the changes we know are inevitable in our designs, developers don't typically factor that learning into our planning or use planning methodologies directed toward identifying and reducing uncertainty.

We adopt Scrum and other "agile" methods, and we adjust our plans as circumstances change, but we generally make no effort to quantify our uncertainty from the outset or to plan in a way that will ensure that we focus on the most important risks.

On *City Conquest*, the DDP system forced us to explicitly identify and quantify all of our major assumptions, build a reverse income statement, and perform a sensitivity analysis of the effect of each assumption on profits. This methodology ensured that we directed our development efforts on the areas with the greatest potential for uncertainty to affect the project's costs and revenues.

But even if you don't use a DDP plan explicitly, the general principle of focusing your efforts at any given moment on those features or areas with the highest value of ((uncertainty) x (financial impact)) can go a long way toward ensuring growth while proactively managing risk.

4. Technology Choices

Technology choices are fraught with risk. There are many recent examples, including *Glitch*, whose developers made clear that the choice of Flash made it impossible to achieve their project goals, and another high-profile case where a developer sued their own engine vendor for technology delays and inadequacies.

Our early tech choices were critical to the success of the game. My own experiences have made me extremely paranoid about the risks associated with licensed technologies and the performance and productivity impacts of scripting languages.

City Conquest places huge performance demands on mobile hardware, including some relatively modest devices such as iPod Touch 4. It required extensive optimization and careful attention to performance at every level.

We chose to build the game with Marmalade precisely because it was not an engine, but only a thin, high-performance cross-platform API layer. It allowed all of the programmers on the team to do 95 percent of our coding and debugging on Windows in Visual Studio using C++, test the game using Marmalade's PC emulator, and deploy identical code to iOS and Android (and possibly Mac and Windows in the future now that Marmalade supports them). Outside of a bit of platform-specific multiplayer code, the initial "first-playable" Android port literally took under an hour.

Marmalade is not without its downsides. Like any tool, it comes with its own trade-offs. The lack of on-device debugging was problematic at times (though this was very rarely actually necessary, as 99 percent of our bugs were our own and were reproducible in the Visual Studio debugger using the Marmalade emulator on Windows). We also needed to extensively customize the resource management.

The use of Marmalade and C++ ensured that we could achieve the critical optimizations necessary to make such an enormously graphically intensive game run at reasonable frame rates on a wide range of mobile devices.

5. Careful Cost Management and Outsourcing

We have seen far too many studios grow too quickly and implode under the weight of their own aggressive expansion. Not only does this hiring dramatically inflate a company's cost structure and the breakeven level required for profitability, but it also usually changes the company's character for the worse. A company's teamwork usually grows much more slowly than its team size, and studios often fail to spread the culture and principles that made them successful to all of the new hires.

Our tiny studio size and careful cost management ensured that we minimized our overhead and kept our breakeven as low as possible. We are far more interested in growing slowly, hiring the right people, managing our risks, and building *City Conquest* and other games into stable, ever-evolving, long-term franchises. We found that our art, audio, and multiplayer engineering contractors produced a surprisingly high level of quality at reasonable costs, and we could not be happier to have been able to work with all of them.

What Went Wrong

1. Mission Design

We knew long before we even began developing *City Conquest* that it's best to teach players gameplay concepts in

small, easily digestible pieces, and wash each one down with a good helping of fun. This is Game Design 101, and as a veteran Nintendo developer, this had almost become second nature, so obvious that it hardly merited a second thought.

Yet somehow, inexplicably, we started by building a tutorial mission that tried to teach every major gameplay concept -- building, upgrading, territory control, dropship pad swapping, game effects, cloaking fields, resource management of gold and crystals -- in a single 15-minute mission with a nonstop barrage of text popup boxes.

After all that work to develop what I felt was a very solid tutorial mission, I received some brutally honest feedback from an external tester. No one wanted to sit through 15 minutes of text popups before having any fun or actually playing the game. And it was just too much information all at once: the gameplay concepts that seemed so obvious to me were of course much less so for the playtesters.

And when they finally did get through the tutorial, they were suddenly overwhelmed with options in the very next mission: the player could immediately build all 20 buildings and use all four Mothership effects before being taught to use them *properly*.

Ultimately, there was no choice but to take a deep breath and refactor the mission design completely. We separated the game into five different conceptual elements -- attacking, defending, expanding your territory with Skyscrapers, using game effects, and swapping dropship pads -- and built at least one specialized mission to teach each of those five core gameplay elements and use them in an entertaining way. We also overhauled to the user interface to allow us to limit the player's building choices to a limited subset defined in the mission scripting, helping us ensure that we would not overwhelm players with options too quickly.

2. Monetization

We released *City Conquest* on iOS as a free download, with a single $4.99 in-app purchase for the full game. The free version contains the full multiplayer experience plus the first five missions of the single-player campaign (out of 14) and the first of the six bonus "challenge" missions.

We had initially intended to release the game as separate "Lite" and "Full" apps, with the "Lite" version as a free limited download. However, discussions with the Marmalade team ultimately convinced us that it would be better to combine these into a single free download, with an in-game paywall separating the demo from the full game experience.

Our reasons for this were well-intentioned: this approach ensured the user only had to download the game once, minimized the potential for confusion between "lite" and "full" versions, made it easier to convert players from free players into full players, ensured that all players could play multiplayer for free, and ensured that achievements and campaign completion were shared across both the lite and full versions.

In retrospect, although we're happy with our conversion rate, it's clear that there are serious limitations to this approach and any others that aren't based on full-fledged monetization. The nature of the mobile market makes it extremely important to do proper price discrimination across the full spectrum of mobile users. There have been several articles on monetization on Gamasutra, and we refer the reader to these.

We are currently nearing completion of the Android version, which we expect to release as an advertising-supported free game.

---

3. Kickstarter

There were two *City Conquest* Kickstarter campaigns. We set the initial goal at a modest $12,000. When it moved too slowly, we canceled it, worked on extensively polishing the game, and returned a few months later with a better pitch, a better product, and an even more modest $10,000 campaign.

The good news is that we reached our funding goal and delivered a much better product than our Kickstarter actually promised.

The bad news is that it probably wasn't worth it. We only got 146 backers and only modestly exceeded our funding goals. The time involved in creating and managing the two Kickstarter campaigns and promotional videos would have been better spent working on the game.

The most problematic issue we faced was our promise to offer our backers the full game for free. At the time, we believed we could use Apple's promo codes to keep this promise, but we unknowingly made it impossible to keep when we changed the plan for our pricing model from separate "lite" and "full" versions to a single combined product with an in-game IAP to unlock the full game.

We'd been assured by several sources that Apple promo codes worked with in-app purchases, so we felt very confident in this approach. We also had a long list of workarounds on hand in case this failed, including hard-coding the recipients' device UDIDs or offering a customized redemption dialog in-game. Unfortunately, promo codes proved not to work for IAPs, and we learned how severely Apple frowns on custom redemption dialogs, the use of UDIDs, and pretty much every single other workaround we'd had in mind.

This was a clear and unfortunate failure of due diligence. We immediately apologized to our backers and offered a refund. The financial impact was minimal since this ended up being only a few dozen pledge refunds at $6 each, but it still should never have happened.

The true value of Kickstarter for us was that we could convert many of our backers into playtesters who provided invaluable feedback. If we use Kickstarter again, we will focus on using it to acquire playtesters and communicate with our audience rather than using it as a fundraising vehicle.

4. Getting Engineering Help Too Late

We realized in May 2012 that our team was stretched too thin to be able to implement and test multiplayer properly. We had a relatively solid first-pass multiplayer implementation in place, but our discovery-driven plan was telling us that multiplayer was a huge risk and required much more attention.

At the same time, multiplayer was non-negotiable: it was an essential part of the game's vision and added far too much value to the project to consider dropping it.

We brought on the Full Cycle Engineering team to replace our proof-of-concept netcode with a serious multiplayer implementation. The multiplayer undoubtedly is a far better experience now than it could ever have been without their contributions. However, we should have recognized the bottleneck and brought the Full Cycle team on board much earlier than we did, and we should have focused on Wi-Fi from the outset rather than experimenting with cellular. The tasks involved were huge and this development effort would have benefited enormously from a much earlier start had we been willing to admit sooner that we needed the help.

5. Due Diligence Failures

Our biggest mistakes were several basic errors in due diligence. None of these ultimately impacted product quality, but we hold ourselves to a higher standard than to be the type of studio that would make these kinds of mistakes.

Other than the aforementioned Kickstarter snafu, the worst due diligence failure was iPod Touch support. For reasons too complex to explain here, the iPod Touch 4 had to be our baseline iOS device, and our discussions with friends and our own research indicated that the iPod Touch 4 was basically the same as an iPhone 4.

This was incorrect. The iPod Touch 4 is a markedly inferior device, and it suffers from much more severe memory constraints. This forced us to bite the bullet and implement a lot of fine-grained memory optimizations. We spent four to five developer-days on time-consuming texture and sound memory usage reductions and major improvements to our asset-loading system and resource management code. Although many of these memory optimizations benefited the overall game, these problems should have been identified and resolved earlier.

We also had problems with a third-party technology we selected for the multiplayer lobby. Very late in the development cycle, we were forced to confront intractable technical issues around this technology, and we had to replace it with a solution based on Amazon Web Services. Rather than selecting this technology ourselves, it would have been better for us to have tasked the Full Cycle team with the due diligence around this and let them pick the best tool for the job.

A third case was our achievements. Some developer friends assured us early in development that there was no limit on the number of Apple GameCenter achievements per game. When we realized that GameCenter actually has a fixed limit of 100 achievements, we needed to drop dozens of achievements to get back under this limit. In the end, this ended up being a good thing, as we only lost a few days' worth of time and we were able to significantly improve the overall quality of our achievements by separating the wheat from the chaff. Although we regret the wasted time and the due diligence failure, the net result was that we ended up with what we feel is truly the best possible set of

100 achievements for the game.

Conclusion

IEDS is a different kind of indie developer, and we are interested in exploring a genuinely different approach to making games. Although we do not claim to have any sort of magic bullet to game development, we are deeply encouraged by what we have seen so far of the results of our approach and the success of our game. We're deeply proud of the final *City Conquest* game experience and looking forward to evolving it and building on its success with the sequel.

Return to the full version of this article