GAMASUTRA
The Art & Business of Making Games

PRINT

Postmortem: Frog City's *Trade Empires*

By rachel bernstein

Frog City was founded in 1995 by Ted and Bill Spieth and myself. My background is in computer science and engineering management. The Spieth brothers are basically game geeks, whose day jobs included being a teacher and a lawyer. The game designers here are focused purely on game design, they don't program or make art. As the company grew, I gave up the lead engineer position to someone who could focus on that 100 percent, so I could focus on keeping us on schedule and in business. So Frog City is a bit different than most developers I hear about, in that we have a shared power structure where responsibilities are divided among several specialists.

Our first two games were *Imperialism* and its sequel. In the middle of *Imperialism II*, our lead engineer went off and wrote a whole new code framework and suite of tools. We started using the new engine for a game which was cancelled when its publisher withdrew from the games business, but it has proven to be of great value for *Trade Empires* and for the new game we are working on now. Having good tools has turned out to be the key to developing a game efficiently.

What is *Trade Empires*?

Trade Empires is an economic simulation and trading game about building civilizations through trade. It is history as seen through the eyes of the merchants. The player is in charge of family trading business. He invests in building markets and setting merchants to work delivering a variety of food products. As the population grows, it develops new and more expensive demands. The player sends merchants to buy resources where they are plentiful and sell them at the populous markets, where they bring in a profit. As time goes by and the people become more technologically sophisticated, they are continually discovering new products and faster, more efficient ways of transporting goods. This presents new challenges and opportunities for the player, but his mission remains the same: balance the competing goals of supplying enough inexpensive goods to keep the population growing, and keeping prices high enough to keep making more and more profit.

The design of *Trade Empires* came in response to my request to the game designers: make a game that would be a good fit for the company's technology and relatively small available team (we also had another game in development), and fit within the budget we were likely to raise at a trying time in the games industry. I'm the person in charge of the schedule and the programming team, while my two co-founders are in charge of game design and the art team. I can always count on them to design games that are fun, deep and engaging, but I added some constraints to the mix: the game should be suited to the engine we had in development (2D sprites on a 3D terrain on the PC), and it should focus on one or two core elements of game play (as opposed to our Imperialism games, which combined turn-based exploration, transportation, production and diplomacy with real-time trade and initiative-based tactical combat). Given that I expected a relatively small budget and short development cycle, I wanted a game where we could focus our energies on making one deep and polished game, rather than getting bogged down with several smaller interlocking games.

Their answer was to make a game about the effects of technological change over thousands of years on the relatively simple model of producing and delivering commodities to make money. The game is broken into scenarios, each of which can be played in a few hours. Their settings range from 3000 BC in ancient Sumeria to 19th century Europe. The types of transportation range from donkeys to wagons to trains, from tiny paddled river boats to steam ships on canals and trans-Atlantic clipper ships. The changes in resources, products, regions and time periods required lots of variety in the art, but the basic code challenges and gameplay challenges remained the same throughout the game. And once the player learns the rules in a simple scenario, he already knows how to play the more complex ones.

What Went Right

1. Finished on time and budget. We actually finished a month ahead of the original schedule, at Eidos' request (They decided they wanted the game out in Q3 instead of Q4. Of course, they couldn't foresee that September 12, 2001 would turn out to be a bad day to ship a game.) We proceeded along the milestone schedule like clockwork. We've been in business for seven years now, and we've never handed in a milestone late.

First of all, since we start building on our own known code framework, it's usually possible to anticipate where the hard code problems are going to be (with a major exception for pathfinding, which is covered in the section on things that didn't go so well). And since the code is fairly clean and habitable, it was easy to pick up and move forward.

Our art team seems always to be on or ahead of schedule. One reason is that our artists are fast. We hire very experienced artists with good academic and professional experience. And we are careful with the artists' time -- we try hard not to ask the artists to repeat work by prototyping interfaces ahead of time and trying to nail down design issues before doing a lot of art work. This is discussed more in the section on how tools helped us develop efficiently.

Our scheduling process is pretty much focused on programming, because that's where the tightest timeline tends to be. The art needs to be basically done by beta, but the coding goes on till the last day. Here's our scheduling process:

1. Designers make design doc.

2. I (schedule/manager person) make a huge set of note cards. Each one has a short phrase on it, could be a sim feature or a technical feature. At first, some of the cards represent tiny day-long tasks, some may represent month-long tasks. I go through the design doc to make sure I cover all the game features, and talk with the programming team to get a feel for all the technical challenges we can anticipate. Programmers' names get penciled in onto each card.

3. I lay out the note cards in long columns on the conference room table. The columns tend to be for game simulation, user-interface, map/engine development, and AI (computer opponents). People stand around the table, rearranging the order of the cards, mentioning things that are missing, breaking down big tasks into smaller chunks, estimating how long things take. Then we start looking for dependencies across the rows, and identifying internal milestones, such as when the major technical challenges are met, and when the game is first roughly testable by the designers.

4. The cards get turned into a Microsoft Project file. Each card becomes an entry, with the cards groups into goals. I pad the programmers' estimates, including my own, since programmers are by nature optimists, but scheduling managers shouldn't be. Figuring how much padding is needed is one place where having been a professional programmer for 15 years comes in handy.

5. Then we're ready to create the milestone schedule for the publisher. The trick is to try to make all the milestone deliverables fairly easy to achieve. In practice what happens is that it's easy to make the first couple, but as time goes by it gets harder and harder, until we're seriously crunching to stay on time. But the freedom it buys us in the beginning allows to rearrange priorities based on what's a most efficient use of internal resources. And during the early part of the project when foundations are being laid is the most important time for programmers to have room to experiment and research a bit. It's important not to waste too much time researching; having the goals and schedule clearly defined helps everyone know how much free rein they've got.

6. Now's when the real work of scheduling begins: the constant maintenance and updating of the schedule. Every week or two we do what's called "This week at Frog City". This is the heart of how we stay on schedule. I sit down with each of the programmers in turn, and they tell me what they've been doing day by day. The programmers also mention things that have occurred to them that will need to get done but aren't already on the schedule. As they explain the engineering challenges they're facing, sometimes we (mostly they) come up with new solutions. And they share insights into how long upcoming tasks will take, with better informed estimates than they had in the beginning. I write it all down in a notebook.

7. If I don't constantly update the Project file to reflect what's happened and the latest estimates of what will happen, it quickly becomes completely outdated and useless. So every month or so, I use the "This Week at Frog City" notebook to update the Project file. I replace the estimates of what will happen with descriptions of what did happen. I compare our estimates to how things went. Usually the estimates are pretty good. Most importantly, I add in all the tasks the programmers have mentioned that weren't already in the schedule. Then I can come up with a new detailed printout for the team of where we need to go and what the priorities are for the next few months.

8. The team meets and discusses the new schedule. I invite people to raise red flags, to pad the estimates, to mention anything that isn't on the schedule. The designers chime in on what order they'd like to get code so they can test and make scenarios.

As a result of this process, programmers always know what their immediate and long-term priorities are. They can pace themselves. They crunch when they need to, and know when they'll have a chance to catch their breath. It keeps us on schedule, and it keeps Frog City a sane and fun place to work.

2. Good code design, flexible code framework and file format We brought on a new programmer three months before *Trade Empires* shipped, and he was making contributions and getting stuff checked off the schedule on his first day. That's because he's very capable and experienced, but it's also a testament to what he calls our "habitable code". The code was pleasant and easy to work with: we used a modular, object-oriented design (being C++ programmers, we like that); the design was based on known abstractions that are familiar and make sense to C++ programmers.

We had developed a flexible code framework before we started *Trade Empires*, and it turned out to be very easy to build a new game with it. It uses a proprietary hierarchical file format, called a Sac. We have tools to convert any kind of data into Sacs, including text from Excel files, art from layered Photoshop files and from 3DS Max renders, fonts, and sounds. We also use that format for scenario files and save games.

There were few dependencies among programming tasks, so programmers didn't have to wait much for each other.

3. Game code was nearly always functional with very few bugs. We try to write code that other people will be able to read and figure out. All the programmers try to avoid acting like parts of the code will be our own private playground where obscure variable names and bad habits can flourish like weeds. Our framework sets a fairly consistent code style, which we all try to follow. Even things that seem minor, like spacing and formatting the code, can be distracting when people use different styles. When different styles creep in, we try to agree on a common approach.

We routinely checked our latest code into SourceSafe, so everyone had pretty close to everybody else's latest code all the time. When the designers would run the game to test and add scenarios, they would get the latest from SourceSafe. Our designers also doubled as our first-response test time, helping us find and stamp out bugs quickly. After the game reached an early playable stage, it stayed fairly robust. We had a very strong beta, and from a bug standpoint the game was shippable after beta. The bug list from the publisher never exceeded a few hundred bugs, which is pretty low considering that text-related bugs get counted in the mix.

Having a game that was solid and playable for more than the second half of its development cycle gave the designers a lot of time to make scenarios and play balance. Having a common code base meant we never had to waste time with big nasty code integrations and synchronizations.

4. Good tool set. Because we started with a good framework and good set of tools, it was easy to do rapid prototyping of art and gameplay. We have been developing and improving the code framework and the tools since before we started Trade Empires, and they helped us move development along quickly and efficiently.

The recurring theme in the development process was avoiding bottlenecks. In general, the more the tools and the code design spread the power around the team and freed people up to make and test their work without needing other people's assistance, the faster and more efficiently the development went. We did a good job with this in Trade Empires, but we're constantly thinking about ways to improve that workflow. A few things really stood out:

The map creation process didn't require programmers. The designers started with digital elevation maps from the USGS for the region in that scenario. They ran some tools to convert it to the kind of map data our engine uses. Then they could use the in-game editor, which was already built into the framework, to export the maps to Photoshop, which they used to fine-tune the terrain and elevation information. And finally they added resources and structures with the editor.

The code framework enabled us to test gameplay quickly. We started with a working engine, so it was relatively easy to reach the point where we could prototype game play. All the game data, such as cost of commodities, prerequisites for technologies, production lines for buildings, speeds of transporters, and scenario setup information is entered into Excel spreadsheets, and then converted into a Sac file.

Information about each scenario was entered into Excel spreadsheets

It wasn't too much work to write the simulation code that ran the economy; it ran on top of the existing framework. And since all the values came from the data Sac files, the programmers and designers quickly reached a point where the designers could test their economic structures. All in all the designers had a good level of control: they could make the maps, control the play balance, adjust the sim data and even some of the rules, all from the tools.

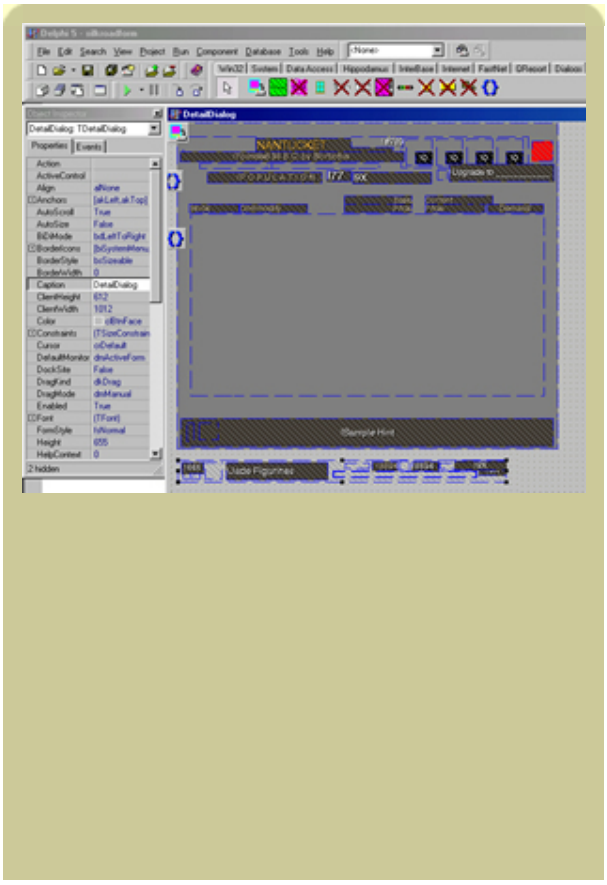
The user interface tools are particularly handy for rapid prototyping. Here's our process:

1. The designers described the functional interface in the design doc, and made a sketch of it (just boxes, no final layout even attempted).

2. A programmer reproduced the functional layout in Delphi. Frog City developed a custom Delphi package that allows us to create View objects and then conveniently set their attributes, such as size, location, position in a view hierarchy, enabled, visible, art, drawing behaviors and font information.

   We developed some systems that allow to indicate a fairly broad set of behaviors from simple scripts. Draw scripts tell how to interpret the art (could be a horizontal strip showing up and down button positions, for example, or something more complicated). They also tell how to draw (draw drop-shadowed text, for example, or condition drawing behavior on state information). Event scripts tell how to behave (eg. act like a button).

3. The art for the user interface was created in Photoshop. Then it was broken into the layers a programmer would use, named to correspond to the layout. The art processing tool extracts those layers, converts them into 16 bit images in the format we like, and puts them a hierarchy of art stored in the Sac file. The art identifier in Delphi would be the path name in that Sac hierarchy. A great thing about this is it avoids duplicating effort: once the artist creates the art, no one has to type in the size or position.



Market detail in Delphi and in the final game.

4. The artist can look at the art using the Sac o-scope Sac viewer, or can wait a bit longer until it's in the game. (Sac-o-scope came even more in handy when reviewing animated sprites, since the artists could run the animations and check to see that the units stayed centered on their spots, and that no frames were missing.)

5. Then it would just be a simple matter of compiling and running to see the interface in action.

For a long time, we avoided using any art from Photoshop altogether. The interface was simply a set of gray boxes on a gray background with black text. It was very easy to throw together. The designers played with the game and requested changes in functionality and layout. By the time we were ready for art, we knew we had an interface that did what we wanted, and we didn't need to request many modifications from the artists. We didn't quite achieve the Holy Grail of doing each interface's set of art exactly once, but we got pretty close.

The localization support was excellent: We stored all of the game text in Excel files also, and processed it into a text Sac file. It was slightly more work along the way for programmers to refer to a Sac pathname than to just type text into the code, but it was worth it. When it came time for localization, we were able to turn about 50,000 word into four more languages in a scant three weeks. That's start to finish, from handoff of text to final testing in the game.

Keeping the text separate from the code speeded development in other ways. The tutorial could be altered on the fly, and the designers could text as they saw fit, without requiring programmer assistance or even a re-compile.

Art integration process was off-loaded from the art team: The data wrangler ran the art integration tools and kept track of art versions, naming, proper placement in the file hierarchy. We have a small but speedy art team at Frog City, and we try to use their time efficiently, focusing on art design and creation rather than data tracking.

5. *Trade Empires* turned out well. We're pleased with the game, but not ecstatic about it the way we were with *Imperialism II*. Some design-related elements appear in this section, on things that went well, and others appear in that not-so-good section.

The core economic model and historical aspects of the game worked well. The game was fun, challenging and engaging. It helped that we have a full-time design staff to really focus on design and play-balance. Between their design doc and their ongoing conversations with the rest of the team during development, Frog City maintained a good unified vision of the game. We also had a unified art vision. Many people were involved in art decisions, from style to scale to layout. There were plenty of discussions, but we all stayed on the same page. Most of the early decisions of art scale worked well, with a notable exception which appears below. The game was very robust and not buggy. We spent a lot of time and effort on the tutorial once we learned how short the manual would have to be, and that seems to have worked well too.

What Went Wrong

And now for the five things that didn't go so well. A lot of things on this list arel are the flip side of items that appeared in the what went welll list.

1. Gameplay would have benefited from some different trade-offs. *Trade Empires* is a good game, but it could have been better. Many of the reviewers liked the game (many three and half star or 8/10 reviews), but raves like the *Imperialism* games garnered were less evident. True, the budget and schedule were a lot smaller and more confining for *Trade Empires*, so we didn't get to lavish the same amount of detail and loving care on it. Even with those constraints, I think we could have made a better game if we had made some different trade-offs.

The number one feature that reviewers missed was more feedback on how the player is doing and on the state of the world. We had a number of features in mind that would have provided this feedback. We designed a separate set of interface panels for charts and graphs. We designed a system to provide rumors of upcoming technologies, warnings and advice from the merchants. Mostly these were features that were absent from the original design, but had occurred to us during development. They would get added to the project schedule in the "gravy area". Many gravy features were added. But these more time-consuming features didn't make it, as we focused our energies on robustness of the code and polishing the feature set we had.

In retrospect, we could have made a better game if we nixed some features that were planned from the start, and added in these others. The number one feature we should have canned was the computer opponents. The game designers realized too late that the game as designed was going to be more fun single-player than with computer opponents. We had three choices at that point: change the design so that

computer opponents would be more interesting (make players own their markets, for example), get rid of computer opponents altogether, or just make the best of it. Sadly, by the time we realized this, it was too late to introduce such a major change to the game design as market ownership. And we'd done too much work in the scenario design that assumed there would be opponent merchant families in the world.

In hindsight, we wish we had recognized the problem a lot sooner, nixed that feature and spent the time instead on other, better features, including more animations for the map (the game we're currently doing has a much bigger emphasis on animations to create a living world).

The game was also lacking the usual multimedia frills that add to the gameplay experience. The game was planned from the start to have a no-frills budget and approach, but as time went on, we starting really missing the frills. We learned a lesson about stinting on multimedia: although gamers may skip through movies and eventually turn off ambient music, these things are crucial and should be included regardless of the development budget. Eidos considered, but eventually declined, funding an opening movie. They did, however, step in with over 90 tracks of music to introduce different regions.

2. Art scale issues. The designers spent a lot of time early on thinking about scale. The conflict became clear: if the map was zoomed in close enough so that the buildings and units would look very cool, then not enough map area would be on-screen at once and the player would have to scroll to see from any market to the next. We thought about having two zoom levels on the map but we realized that we'd end up with the nice zoom level for screenshots, and the zoom level that was actually good for playing on. We didn't have the time to support both at that point (we have multiple zoom levels in the new game we're developing), so we opted for the better level for gameplay. But it was a shame that so many of the cool details in the art couldn't be clearly seen.

It's a shame that so many of the cool details in the art can''t be clearly seen.

I think the designers and artist made the right decision, given our budgetary constraints, but where they got it wrong was in picking a size for the production buildings. Early on, the designers were worried that if the production building footprint was too large, market areas would quickly become congested. It turned out that concern was misplaced. We could have let production buildings take up four times the space we

chose, and then we could have used a much more detailed approach to their art that would have done a lot to promote the feeling of a real world.

3. Beta testing: too little too late. If we had done more beta testing, with a larger group and earlier on, we would have gotten the kind of outside feedback that would have helped us realize that some of the tradeoffs we were making were going the wrong way. We didn't miss some features nearly as much as other people did -- we were so used to the game that we adjusted our play style and underestimated how important the missing features were.

We did out best to show the game to as many veterans and rookies as we could get into our office. But just not enough eyes saw *Trade Empires*, especially without having a Frog City person at their elbow to explain away any possibly ambiguous elements of the interface or game rules.

4. Didn't do enough to get the marketing people excited. Our approach to interface design meant that for a long time, there was no interface art to look at. The interface was a bunch of functional gray buttons and text boxes. This made for very efficient use of the artist's time, but it didn't do much to excite marketing people at the publisher. Marketing people have many projects to juggle; they shouldn't be expected to imagine art in place of gray boxes. In retrospect, time spent making mockups of the interface and special hi-res images just for the marketing teams would have been time well spent. As it was, the game flew below the radar during a time when the publisher was getting increasingly shy of PC games market. So it never got any marketing attention. Getting the support of marketing people is crucial: not only do they control the advertising budget, but their support rallies additional interest at the publisher - useful for things like extra multimedia, bigger manuals, more attention and thus more feedback from more perspectives during development. They not only help the game get in front of paying players, they can also help make the game better.

5. Seriously underestimated time to write pathfinding code. Pathfinding was the number one mistake in our scheduling. Over and over we failed to realize how much time the game would spend pathfinding, and how much time we would spend writing the code. We didn't realize that many things which we thought of as separate challenges were really all pathfinding and should be designed in a comprehensive way: creating a path for units who already know their destination, having the unit sprites follow the path and stay centered on their pathway art, constructing the pathways, drawing pathways.

And then there were the computer opponents. For a long time, we thought of that block of code as simply having to do with strategic economic decisions: what should a merchant buy, how much, where should he sell it. We didn't realize that the hardest part would be having a merchant figure out how to get from point A to point B, especially given that much of the time he would need to construct roads or ports and buy specialized transporters to get him there. We scheduled in the economic decisions, but overlooked the bulk of the AI programming job, which was various forms of pathfinding and path-construction. In particular, the hierarchical levels of pathfinding (multiple regions and multiple pathfinding contexts depending on required pathways) were not generally organized.

Canals, railroads and rivers in industrial England

Trade Empires is a game about networks of pathways. The game spends a lot of time in the pathfinding code, so it needs to be optimized,

and it needs to play nicely with the rest of the code and share the CPU time. Even as we realized new pathfinding challenges, we consistently underestimated how long they would take to write. We did end up with several excellent and reusable code modules for pathfinding, but now I'm careful to add a lot of padding to any time estimates involving pathfinding.

In addition to doing a better job estimating how much time writing the pathfinding code would take, I wish we had recognized how a couple of features were making the whole job much harder, and changed the game design. *Trade Empires* has several types of pathways. Merchants can travel over trails, roads, railways, canals, rivers and oceans, or they can bushwack on mules and donkeys with no pathway at all. We should certainly have gotten rid of the ability to bushwack without a pathway-it would have simplified the pathfinding challenge a great deal. We tried to make consistent systems so we could re-use the pathfinding, pathway construction and pathway drawing code. But the rules were just too different. Canals weren't handled until fairly late, and their scale and drawing rules were different from everything else. All the time that was spent supporting special-case canal code could have been saved if we had planned a more consistent and comprehensive set of rules from the beginning.

Conclusion

Lots went well during the development of Trade Empires: several publishers were interested in the proposal, we worked with great people at Eidos, we finished the game ahead of schedule and on budget. Though certain parts of the development proved more difficult than we had anticipated, and some design decision might have been made differently if we had it do over again, all in all the development process was very smooth and efficient and company morale was high throughout the project. Most importantly, we made a game we are all proud of.

Game Data

Trade Empires

Developer: Frog City Software, San Francisco, CA

Publisher: Eidos (Take Two Interactive outside the Americas)

Team Size: 9

Number of Contractors: 3

Length of Development: 15 months

Release Date: September 12, 2001

Platforms: Windows 95/98/2000

Development Hardware: Various PC systems, from 400Mhz Pentium IIs to 1 GHz Pentium 3s and 1 GHz Athlons. Assorted nVidia graphics cards, plus some ATI, Matrox and GeForce cards. Hard drives ranged from under 10GB to 60 GB.

Development Software: Windows 98/2000, Borland Delphi, Adobe Photoshop 5.5 and 6.0, 3DS Max 3.1, SGI_STL, Miles Sound System, Microsoft Visual C++ 6, SourceSafe, Word, Excel, Project.

Proprietary Software: *Pork* (process assorted types of raw data including layered Photoshop files, animation frames from 3DS Max, truetype fonts, sound and music files, and Excel files). *AXVI* engine and code utilities, *Smalltime* animation compiler, *Hippodomus* user interface Delphi package, *Sacoscope* file and animation viewer, Production Artist file renamer, *FrogLok* art version control

system, *RICE* console system.

Notable 3rd party Technologies: DirectX, Miles Sound System.

Project Size: roughly 350 files (C++). 32 animated units, 500 buildings, 4 sets of terrain textures and foliage art, 50,000 words (localized into six languages with more underway).

Return to the full version of this article