Aquarius Postmortem

Thalassic Games

July 6, 2003

What Went Wrong

We didn't have time to allow the whole team to review and refine the final TDD – several flaws in the technical design could have been identified early on and suggested fixes made. This would have increased productivity late in development.

If we had had time to finish the GDD and TDD in the summer then we would have had several extra weeks of coding – allowing for the implementation of some more higher goals and polishing of existing features.

Failure to formalize source control procedures (file and directory naming conventions, directory structures, and source control usage procedures) caused some thrashing. On a related note, since not every team member knew how to use VSS correctly, there was some unnecessary suffering and lost work. Also, some members of the team just don't like VSS. In general, it's very important that every team member that needs to use a tool knows how to use that tool correctly. Furthermore, if two or more team members are to be directly collaborating, it may be necessary to create some formal sense of "correct tool usage" to ensure that the team members don't step on one another's toes during development.

The style guide should formalize interface naming conventions, minimization of link-time dependencies, self-containedness of header files, correct precompiled header usage, and whether .h or .hpp is the correct extension for a header file. Some feel that it is inappropriate to send text messages to the compile window. Some feel that a variable naming guideline (noun-then-verb, for example) would make identifiers easier to find. Finally, the style guide must be short and terse (1-4 pages). These measures would have reduced build times, increased code-style consistency, and reduced thrashing.

Long compile times were a problem. In addition to the measures taken

in the style guide, we should consider the future use of libraries once interfaces are finalized (as an added bonus, libraries could help enforce self-containedness of modules). If libraries are made an integral part of the development process, streamlining the process of building a library would be desirable.

Using two compilers caused a little pain. We should have made the team decision as to whether or not to use two compilers. If yea, we could have systematically dealt with the two compilers' idiosyncracies across-the-board. If nay, sticking with one compiler would have meant we'd only need to accomodate the weirdnesses of that particular compiler. As it was, we were relatively lucky to have gotten away with using VC6 and BCB6 – next time, compiler idiosyncracies could prove to be more destructive.

There was too much voluminous, overly detailed, legal-style text in the GDD – it took too much time to read and understand. The UI was clearly the worst offender: flow-charts, diagrams and RAD prototypes are vastly preferable to reams of text. An interactive, RAD-created demo is worth ten thousand words, and each screenshot worth a thousand. On a similar note, the UI design should have been written at a higher level (maybe even as high as simply describing what the player can do at a given point in the game) so that game documentation maintenance – even in the face of significant changes in the UI design – is efficient. Much of our UI woes stemmed from the simple fact that our user interface had to be quite complex.

In general, there was some feeling that documents were too large and should have been split up into smaller, more granular documents (for example, a UI design document, a "sample-turn-fluff" document, a core gameplay document, etc) so that each is more focussed, less intimidating and, hence, more useful.

Certain sections of the TDD were too low-level to be useful to anyone on the team except for the writer of the given low-level section. Next time, low-level design sections should be separated into small "personal" documents, leaving only the high level design – the stuff that everyone on the team should be aware of – in the TDD. For example, we should just use "generic containers" in the TDD to improve brevity – implementation details like which datastructure will be used to hold pending message packets and which algorithms will operate on this datastructure should be left in a separate, "personal" document. There was some feeling that bullet points in TDD were more readable than long endless paragraphs.

The Networking and Messagehandler modules should have been broken up into more .cpp/.h modules up front for better code organization. Networking did not manage memory well – the game can't run for more than

about half a day before memory leaks defeat it. Next time, memory management should be prototyped and proven before being designed around. Macro hackery should have been used to create code that is compiled only in debug mode – this would have made it much easier and less error-prone to perform clean error-checking in the debug build that is optimized away in release.

Building a good independent testing framework for every module is part of the software-engineering process and needs to be planned for from the beginning – our testing frameworks were incredibly useful but were also a cause of frustration as they broke every couple hours due to others' updates.

There was some feeling that the second semester scheduling wasn't as good as the first – more on that later.

Some felt that being split across two rows hindered communication – but being elbow-to-elbow is also felt to be suboptimal as well. Some missed the lack of windows. No fix for this in DigiPen, but in the real world, people get offices.

Click tracking the soundtrack wasn't rigorous enough – the interactive soundtrack failed. Next time, outside musicians should be required to play along with music tracks that are known to be perfectly in time.

There was not enough development time. This lack caused John to ditch two classes, first playable to come along much later than we'd have liked, and feature lockdown to never occur. Team time was not entirely realistic – our scrounging for a few spare hours was only partially successful, since frequently team members would arrive very late – which, of course, castrates the focussed productivity team time is supposed to provide. Finally, when it became apparent that Matt Phillips could not work on the project, some felt it would have been appropriate to re-evaluate the design of the game to narrow its scope.

Code reviews would have been great – it likely would have propagated knowledge throughout team, helped individuals to kick bad habits and get new perspectives on their code, but even having one other team member read through another's milestone codebase on a regular basis would have been inconceivable given our time-frame.

A formalized high-level testing procedure late in development could have kept the game more bug-free and more fun, but even having the team test all major features on a weekly basis was not reasonable given our time-frame. Finally we didn't have time to really test John's AI System hypothesis to see if it does good, general AI – we were working on core features right up until the end.

Another development approach – that is nearly impossible to practically

implement at Digipen – is to do a very small, high level GDD, then have coders prototype the technologies that are the most core and the most difficult, and then base the game on those prototypes. In the real world, it was agreed that one needs at least nine months to allow for 1-3 months of prototyping, for a game of modest scope. Finally, the consensus was that prototyping gameplay (possibly using middleware) made for a more fun game, and prototyping technology made for less risk come production time.

What Went Right

The team. As individuals and as a group, we were highly motivated, cared about our work, had good communication, and had pretty compatible work styles – we figured out what we were going to do, how we were going to do it and then stuck to plan.

The team time that we did manage to achieve was very productive.

We got experience in taking the design-bible methodology to its logical extreme, and came a long way in knowing how much (and what kind of) documentation is actually useful. In a nutshell: much less game design documentation and much higher level, more flexible game design documentation.

doxygen worked great – it provided good documentation and enforced a good style of commenting. John Corpening uses it professionally.

John Corpening got both the AI and UI core features done – exemplary, given that he was essentially doing the job of two.

Higher goals allowed us to painlessly cut features as necessary and still accomplish all that we could – we still got the core design and then some done, even without Matt Phillips. As an example of this, the non-interactive music worked out OK – we successfully used a backup plan after the interactive-music preproduction ambitions failed.

Supportive "Publishers". Both Larsen and Moore, almost without exception, stayed out of our hair and let us work our own way. And they both gave us As!

Object-oriented technical design was successful. We thought the design through, and that up-front thoroughness paid off when integration was smooth and relatively painless. In particular, the command pattern (which provides a specialized interface wrapper around an object) worked well: the graphic client, tank client and sound client didn't bog down interfaces, were clean and easy to use and supported encapsulation and code organization.

The MessageHandler succeeded in hiding Networking from the rest of the

game system – it could have been even better had we used command-pattern clients to improve code organization

Successfully learned about software engineering and software design.

Successfully learned about AI. It seems as though the theoretical concepts of machine learning are a deep introduction to AI in general – John feels that Aquarius specifically helped him do better work professionally.

There was some feeling that the more detailed scheduling of the first semester was successful. Detailed schedules can help give sharper project visibility and is very important early in development when task dependencies can slow people down. Therefore, schedules must be at least detailed enough to prevent task dependencies from becoming an issue. On a related note, task dependencies should designed and scheduled to work out at the highest level of scheduling possible to ensure good modular design. One suggestion was to have two layers of scheduling – a general schedule and more specific, fine-grained schedule to support those who prefer to work with either type of schedule.

Graphics turned out much better than expected – the game looked pretty good!

Got experience with VSS (a good thing, since VSS and CVS are industry standards).

Didn't succumb to the temptation to change tools midway through development (from VSS to CVS), which almost certainly would have been far more trouble than it was worth.