Creating a Data Driven Engine
Case Study: Age of Mythology

Robert Fermier
rfermier@ensemblestudios.com

Abstract

Building data-driven systems into a game engine can help extend the power and flexibility of code, reduce programming dependencies, and empower content creators. Careful attention to ease-of-use, simplicity, and knowing which problems can be handed off to non-technical personnel is key to success.

Data-Driven?

What does it mean to be data driven, anyways? All systems run on data in one form or another. However, a system which merely uses data is quite different from one which is designed to be truly driven by that data. Roughly speaking, a data driven system attempts to adhere to a few principles which ordinary systems may not:

- **Division of Labor**. In a data driven system, there is a strong emphasis on separating the creation and maintenance of the data from the underlying code that uses that data. Programmers can maintain the code, while content creators maintain the data.
- **General-Purpose Functionality**. A data-driven system does not merely set out to solve a given problem; it instead seeks to solve all problems of that same form. The original problem then falls out naturally as a result.
- **Modularity**. When creating a data driven system, it is important to support not just a limited set of features that are known to be needed, but also be able to easily extend the system to support other features of a similar nature.

Advantages

Any software developer is familiar with these basic concepts and their benefits in a general sense. However, when the reality of game development intervenes, they are often brushed aside to meet more tightly-defined short term goals or just to reduce the scope of the problem. During the development of Age of Mythology, we tried to take a long view as much as possible and really stick to the principles of creating a data driven game engine. There were many positive results that emerged, making an engine which is flexible and extensible, as well as creating a more efficient process for game development.

Having a very flexible game engine is the first and foremost benefit of being heavily data driven. Rather than having a vast number of individually specialized systems that would have to be coordinated and developed individually, we focused on building fewer systems that could handle a wider variety of uses. Flexible systems could sometimes be used to solve entirely new problems without any additional code. Designers (or other content creators) can experiment with the bounds of the systems that have been created, and often use the tools in ways that were never even considered when they were first built. Even more powerfully, once you start to have a sufficient "critical mass" of these systems that can be highly customized, the fact that they can be coupled together causes a combinatorial explosion in the effective power of your system. The ultimate expression of this concept would be being able to create entirely new games, or even just radically change your game, without any changes to the code.

However, changes to the code are not only important, but inevitable. Being able to easily extend existing systems is critical to adapting to the rapidly changing environment of game development. If the path of least resistance for adding a new feature is to build upon an existing system, developers will go that way rather than trying to reinvent the wheel. Having such an infrastructure in place can make those changes easier, both for the programmers writing the code and also the designers or artists using it, since they do not have to learn something entirely new. It also means that when you build tools for a system, such as a debugger for a scripting system or an integrated help system, you can reap the benefits of those tools for your newly added feature as well.

Creating a game is not just about the programming, of course. Coordinating the efforts of the people adding a new feature with the people using that feature is a great deal of work, and is often vastly underestimated in planning for development. By shifting as much as possible off of the code and onto the data, typical development process problems can be eased significantly. This allows designers to tweak the game themselves as much as possible instead of requiring programmer intervention. Changes can be seen with instant or at least very rapid turnaround, instead of requiring a more lengthy development and build generation process. The very process of defining what those systems are is in itself an aid to communication. Prototypes can be generated directly by content creators, and programmers can focus more on their code and less on issues specific to the data itself.

Disadvantages

There are hazards, of course. In addition to the obvious consequence that being more powerful and general involves more up-front investment in work, there are a number of less-obvious problems that we encountered during development. Sometimes the systems created were too powerful for those expected to use them, or the resulting systems would be a hacked-up behemoth understood by

no one. There is also a temptation to over-engineer problems that are honest special case systems requiring more specialized code. It can also change some traditional boundaries of responsibility, causing confusion and miscommunications.

Having a system be too powerful may seem like a silly notion, but it can be a very real issue. Most users, when faced with a system more powerful than they are comfortable using, will either seek help or only use it in a very conservative fashion. Some, however, know just enough to be dangerous, and can use that knowledge to wreak havoc without intending to, or even being aware of it. Powerful systems are generally complex, and there is often a wide variance of technical ability among non-programmers. This can result in more time spent teaching the system than it would have taken to simply implement the feature directly in code. Creative skill does not always coincide with the kind of analytical skill is required to use a technically complex system, especially one without proper support and tools.

Even when users understand how to use a system, they don't always use it in the best fashion. What may start out as an elegant data-driven solution may turn into a Rube Goldberg nightmare of impossible complexity. Suddenly it can take days to make even simple changes, if anyone feels brave enough to even try and change it at all. It is very important to keep in mind the importance of the "keep it simple, stupid" principle. Rather than ask a programmer to add one simple feature that would take ten minutes to add, a designer might spend weeks making a content-side solution. This is the development equivalent of swatting flies with a nuke. While it may get the job done, it's hardly the right tool for the job… and the fallout may kill you later.

On the programming side, many dangers abound as well. It can be very easy to create an over-engineered solution to a problem, solving a case so general that it requires a vast infrastructure when in practice a much simpler approach would handle the most important 80% of the problem. An infinitely powerful system still isn't very useful if it can't be practically applied to anything. Similarly, if a vastly powerful system is, in the end, only used for one specific thing, then all that extra work was wasted. The end-user doesn't see all the power that went untapped, but they definitely will see the bugs or schedule delays that may occur from trying to maintain an over-engineered system. It can be very difficult to accurately assess the problem while in the middle of building such a system.

While data driven systems can be useful for solving some kinds of communication problems, they can also create new ones. It is vital that the programmers understand what the designers are trying to get the system to do, and that the designers understand how the system actually works. Without a tight communications loop, it is easy to get into a spiral of misunderstandings. There are also just practical questions that crop up, such as "who fixes the bugs"? When programmers spend time finding bugs that prove to be incorrect

data, they can get frustrated, and when designers spend time trying to tweak numbers on a system with a fundamental error they can get equally upset. While these sorts of communication problems are hardly unique to data driven systems, they may become much worse as a result.

Case Study: Age of Mythology

To more specifically illustrate the risks and rewards of creating a data driven engine, it would be useful to look at a more detailed case study of development. Throughout the development of Age of Mythology, a great deal of effort was put into creating strongly data driven systems that would put as much power as possible into the hands of the designers and artists, or even just allow programmers to tune a system without having to change any code. By seeing what worked well and what was a horrible failure, it may serve to help understand the issues involved.

For those who may be unfamiliar with it or its development history, here is some basic information about Age of Mythology and the engine created for it. Following in the footsteps of the highly successful “Age of Empires” and “Age of Kings” games, we first and foremost had to build a very solid RTS engine, capable of all the gameplay, functionality, and polish that those games possessed. As we were starting from a clean slate, with no legacy code, we had a great deal of basic infrastructure and system building to do just to catch up to where we were before. Of course we wanted to exceed that original code base, not just in functionality, but in the robustness, cleanliness, and overall quality of that code.

Unlike its predecessors, Age of Mythology would be entirely in 3D, taking full advantage of modern hardware acceleration and the many benefits that 3D provides. However, we weren’t willing to make any compromises in the final experience that the end user would have. So Age of Mythology’s 3D engine had to deliver a graphical experience that was just as lush and detailed as Age of Kings’ highly acclaimed 2D graphics, and we couldn’t sacrifice any of the ease of use that users expect from the series. This would be a great undertaking, and we wanted to make sure that the work we were doing could be extended for additional games in the future.

The engine for Age of Mythology has many different data driven systems. Rather than go into great detail on their individual implementations, successes, and failures, I will present the high-level issues and the theory of how it was intended to work in a data driven fashion. The theory will then be brought into reality with some analysis of how the systems actually worked in practice. There are many data driven systems in Age of Mythology, but the largest ones are worth focusing on here, such as the development infrastructure, the UI, the scripting language, the game database, and the animation control system.

Throughout all of these decisions, there are a few common tools or recurring issues that had to be addressed, that are worth noting specifically. Almost all systems relied on some kind of external data file, but the roughest systems generally relied on direct editing of those files. In almost all circumstances, it was of high importance to keep those files as "human readable" as possible. We went through many iterations of format, but eventually settled on XML as a general standard, which helped that considerably. Where possible, we tried to create in-game tools to edit the files, but the time needed to create those tools often was never found. In a few cases, powerful standalone tools were created just to edit the data, such as our database and particle systems. Figuring out who will be using such tools, and when you can get away with the cheaper options is an important decision to make.

The Config System

Age of Mythology's configuration, or "config" system, is one of the most basic, low level systems in the entire engine. It is a simple system for storing user-specific information, somewhat analogous to an .INI file or the registry. However, we wanted to have a more centralized system that we could tinker with and extend ourselves. The basic goal of the system was to make it as easy as possible for programmers to create new config variables and use them in their code, so that the system would get widely adopted. So for example, a programmer could write:

```
if (gConfig->isDefined("myFunkyConfig"))
      Blah();
```

That new config variable could then just be used, without having to create any further new infrastructure. Being able to just pluck new variables out of "thin air" would make the system as easy to use as possible. In addition to the "plaintext" system, a programmer could also go one step further and create a "formal" config, which required slightly more code to create, but allowed O(1) speed lookup of the variable, as well as some other niceties like associating on-line help info for the variable. Even formal configs were as lightweight as possible, and could be created by adding a single line into a table, like so:

```
{cConfigMyFunkyConfig, "myFunkyConfig", "here is some help text." },
```

Age of Mythology has a great deal of network support built in, so the config system could also understand that some variables had to be forced to have the same values for all players in a multiplayer game, for example. In addition, some configs could be specified as "persistent" values that stayed with your user account after you logged off, and could even be retrieved from the server when playing online, so that you would have your personalized settings no matter which machine you might be playing from.

Config variables could be set from any number of sources, ranging from the server's persistent user data to the command line. In-game tools were created so that all the config data set for a session could be browsed, filtered, and changed. The whole system was designed to be very generic, just storing information with an arbitrary name, so that it could be used by all the other systems in the game.

In practice, the config system worked out fairly well. As a very generic system, it was successfully used in a very wide variety of systems ranging from the terrain renderer to the pathfinder. Once everyone on the development team became more familiar with the config system, it was much easier to add new variables and just expect that anyone who wanted to use it could do so. Editing config files was very easy, and they could be used for both user-overridden values as well as "official" data values. Perhaps the best and most interesting use of the config system was that it could be integrated very smoothly into UI elements like menus or hotkeys. Even some larger pieces, like the in-game options screen, were created almost entirely with the config system.

A number of problems cropped up with the config system as well. The worst of these was that the easy-to-use plaintext config variables were in fact, **too** easy to use, and found their way into very performance sensitive locations. Several large passes of removing plaintext configs and replacing them with formal configs *en masse* had to be made. There were also a number of problems as a result of how many different sources could set config variables, and it could become difficult to sort out the tangled mess to figure out where a given value was actually coming from.

The Console System

Having an in-game console is a standard tool in the FPS world, and seemed like it could be of equal use in an RTS engine. Console commands could easily be associated with C++ functions, and it provided a useful generic "bridge" between the run-time environment and arbitrary tools and utilities.

New console commands could be added fairly easily, by providing a C-style interface and using some provided macros. Help for the command, as well as help for the individual parameters, could be specified right there where the command was declared, making it much easier and thus more likely that the help would actually get provided. Tab-completion and various command search modes were created in order to help other developers actually find out what commands existed and how to use them.

The console system used the game's scripting language, "XS" (for e**X**pert **S**ystem), as the actual mechanism for converting the raw text commands into parseable parameters and actual C functions. This provided some basic type checking, syntax, and type conversion features "for free".

Using the universally accepted, ANSI-standard tilde key, the console dialog itself was designed both for easy input of console commands, as well as tracking logged output from the console commands or the game itself. The dialog maintained a history list and was scrollable and resizable.

In practice, the console system was probably the single most successful infrastructure system built for Age of Mythology. Almost every programmer added many useful commands and used it extensively for testing new features or debugging. The online help features were used heavily, and it rapidly became a staple of development. There was good synergy with the config system, and using console commands to set config variables became very common practice.

The single most important thing about the console system was how tightly it was integrated into the game. Virtually every system in the game had its own set of console commands, which were at least moderately documented and used. Some entire systems, such as the hotkey remapping system, could operate off the console system alone. It became a vital linchpin of the UI system, a topic which is discussed in more detail below.

No system is complete without some problems, of course. The syntax for the console commands, while fairly unrestrictive, still required precision in where parentheses, commas, and the like went, which caused a lot of problems with very non-technical users. Even among more experienced users, it was unavoidably easy to make a typing error and have things not work.

Another problem that cropped up was somewhat more subtle. Because console commands were so powerful, they could be easily used to either cheat in a multiplayer game or at least make one user's machine go out of synch with the others. While all of these commands were suitably disabled for security reasons for "ship" builds of the game it remains a continuing effort to make sure none slip through the cracks or are added without appropriate security.

The UI System

The user interface system in Age of Mythology, which covers everything from the on-screen GUI to the hotkey system, was ultimately one of the most data driven systems in the entire engine. A great deal of programmer time was always historically spent on UI creation and maintenance in the past, and it would be a tremendous asset if that process could be streamlined so that the UI designer could directly make modifications or even create entirely new UI content without needing any programmer involvement.

All of the actual GUI screens in the game were created with an XML UI layout specification system. The UI designer could create new entries in an XML file, and those would directly correspond to a new button or on-screen element.

Visual and behavioral elements of the UI could be specified through the XML file as well. When new functionality was added to the underlying code that created the GUI, it could easily be exposed through the XML to be used from the data side through a simple DTD mechanism, which is how XML formats are specified.

The various elements of the GUI were all designed to couple tightly with the other low-level data driven systems in the game, such as config variables and the console. For example, menu elements or edit boxes could trivially be set up to adjust config variables, and that formed a natural and easy mechanism for programmers to create a system and then allow the designers to independently hook up a control mechanism for it. The entire options screen in Age of Mythology, for example, is powered entirely by having the XML layout for the various controls poke config variables for things like sound volume, and the like, thus requiring no specific “options screen” logic hookup at all.

Providing key mappings, an important part of user interface for both the development team and for the end user, is all handled through console commands. Once a console command existed to provide a mapping between a given hotkey and a console command, the key configuration files were then just nothing but a list of those console commands. That architecture fit very neatly into existing tools for automatically handing off lists of console commands to the game to execute at startup.

One of the most powerful data driven features of the UI system was how the console was used as a universal control mechanism. Every user input in the entire game was transformed into a console command via key-mapping style console commands or GUI elements. Whether it was clicking on a button, hitting a hotkey, or even making selection in the world, it was broken down into this one common model. For the GUI alone, this meant major new functionality could be specified entirely through data.

As the body of console commands grew throughout development, the power expressed through the GUI system increased automatically. Because hotkeys went through this same system, it was very easy to automatically provide hotkeys for anything that could be hooked up in the GUI. Even all the gadgets created in the system were themselves given a “human readable” name, and could be accessed via the console commands. This allowed the system to easily control itself through designer-provided data. It was a fairly common construction to have GUI elements like buttons know how to reveal or move around other GUI elements, all through their data.

By adopting the “universal console” model, it transformed the UI into a dispatching agent, converting user inputs into the relevant matching console command. While this may seem like a minor distinction of how the system should work, it proved to make a major difference in where the coding time was spent on the UI. Rather than spending time moving buttons around, or even

creating UI screens at all, effort could be focused on adding functionality to the core UI system.

In addition to data-driven input control, the output elements of the GUI could be customized as well. The game provided a "data fetching" mechanism, an extension to the text format markup system that allowed it to actually get game variables that were updated in real-time. So if the designers decided to add an area to the UI to show the unit's hit points, they only had to embed a format string like "{currentUnitHP}" into the text for that GUI element.

Ultimately, the system succeeded in improving the process, moving a lot of data tweaking and layout work off of the programmers and onto the UI design team where it belonged. While there was still a lot of programming work to be done creating more sophisticated custom interfaces where needed or feeding data from the game to the UI system, it allowed the work to be focused on a higher level.

One of the biggest improvements in the UI process was how it enabled rapid prototyping of new screens or UI changes. Our design process for the UI is very iterative in nature, so this allowed the UI team to try out a lot of things with very short turnaround. In addition, where programming work was required to hook up more complex functionality, having a pre-existing UI shell to work off of helped the process considerably.

Early in Age of Mythology's development, there was no layout tool at all, and everything in the UI had to be created "manually" through code, in a more traditional fashion. Because the system was designed to have a wide breadth of functionality, without a good set of tools to help use the system, it proved extremely unwieldy. The turnaround time for creating a new UI element went from several days in the old system to a couple of hours in with the help of the XML layout system. When it was implemented it entirely changed the perception of the utility of the UI system.

The XML format was ultimately a successful part of the system as well. It enabled the UI designer to use off-the-shelf XML editing tools which could provide a nice editing environment without any developer intervention. The DTD format specification file formed a rudimentary kind of documentation as well, enforcing syntax constrains on the XML file that would have been much more difficult to maintain in a more freeform text format.

One real downside to the system was the very steep learning curve that it demanded of its users. Even with the assistance of XML editing tools, having to understand a structured format like XML was daunting for non-technical users, and a real problem for adoption of the system. While the system was very powerful, it was so open-ended that it could be extremely difficult to figure out what to do. Despite the ability to fully customize the UI in data, it often required

programmer involvement to provide the “magic” strings of console commands or data fetcher formats to get the UI set up properly.

The whole system was held back greatly by the lack of a visual layout editing tool. While it would not have been technically that great a hurdle to create a simple 2d app that would output the right XML, the time was never available in the schedule. This was inefficient for the designers that had to use the system manually, doing layouts in tools like Photoshop and then laboriously typing in coordinates. Even that process wasn’t too bad for initially creating screens, but it made maintaining them a true nightmare, since moving elements around was error prone and tedious.

While having a strong data driven concept like the “universal console” at the heart of the UI was very powerful, it had a lot of resource costs as well. The constant act of converting input to strings to be interpreted as console commands was not fast, although user inputs happen infrequently enough, compared to rendering and running a simulation, that it was rarely a problem. A more common problem encountered was that storing off all the command strings, GUI names, and the like caused the system to use up a lot more memory than was expected.

In retrospect, Age of Mythology would have probably benefited greatly by taking all the GUI elements that were not a part of the game proper, such as the scenario editor, and just building them with the traditional Windows GUI. A lot of work went into recreating the functionality in the Age of Mythology GUI system that already existed in Windows, which was only used by the scenario editor. If we had made that call at the very beginning of the project, we probably could have gotten our editing tools into the hands of the content creators much earlier.

Scripting Language (XS)

The XS scripting language originally was designed only for the computer player AI in Age of Mythology. As we began to build the infrastructure of the game, we discovered that there were a number of different systems where a more procedural, programming language type system was required, and XS was extended to meet those needs as well.

As mentioned earlier, the most basic functionality of a scripting system like XS is to provide a “bridge” between text data and actual C++ function calls. New functions in code could be exposed through a series of macros and directly callable from the game. XS provided hooks for parameter types, syntax checking, and a help system. Because it was so easy to add new content to XS, a lot of content was generated for it in a short amount of time, and all of that content could be used between all the XS powered systems.

While the console system just used XS for simple calls, XS was created as a fully capable C-style programming language, including functions, variables, conditionals, and looping. The AI designers used XS to create sets of "rules" that could fire with varying frequencies, and those rules governed the behaviors of the computer players. XS was also used to provide the procedural rules governing random map generation, as well as controlling all the cinematics and special triggers for the single player campaign.

Scripting languages are not a new or revolutionary idea in game development. There are a number of approaches that do not involving building everything from scratch, ranging from pre-existing languages such as LUA to actually using C++ DLLs. For Age of Mythology, we opted to create our own system for a number of reasons. Aside from the general desire to be able to have full control over all the systems in the game, writing our own system allowed us to integrate a lot of tools and find more unorthodox uses of the system like implementing the console commands.

At a higher level, it was logically important that we have control over the very syntax of the scripting language. While the XS format is very C-like in order to make it easy for programmers to use, it was vital that it not expose the full breadth of power that a full programming language can use. For example, the looping construct in XS supports only a limited iteration-style syntax, so that it is impossible to make infinite loops or have common looping logic bugs that are very common to inexperienced programmers. By definition, the people most using XS as a scripting language would be non-technical users with only basic experience with practical programming. So accordingly, we needed to be able to take every precaution in XS to make that as easy as possible, and no existing language offered that level of safety.

Using a scripting language to control the AI was a well-proven architecture from Age of Kings, so we had no doubt about its benefits. Being able to leave the AI designers in control of the high level strategic flow of the AI keeps it focused on the gameplay, leaving the code to handle all the myriad details of the execution of that strategy. XS was designed with the AI in mind, and in practice it certainly was the system which most provided the most volume of content and also the most stressful performance conditions for the system.

One tool of specific note that was created for XS is the integrated debugger. This was a full, source-level debugger with symbols, breakpoints, and all the trimmings. Having a powerful tool like this was tremendously useful for debugging complex logic in the AI. It could also be leveraged to help all the systems that used XS at their core.

Random map generation has gone through a significant evolution throughout the Age series. In Age of Empires it was handled largely in code, while in Age of Kings we moved to a declarative system. For Age of Mythology, the system

used the full procedural power of XS. This allows for much more complex and inventive random map types, although it does put more of a burden on the creator of the maps because whole new realms of bugs can occur in the logic of the system, not just the data. Using XS retained many of the core benefits of the random map system, making it very easy to rapidly test a wide range of changes, and allowing end users to generate their own powerful custom random map scripts.

The way in which the triggers and in-game cinematics of Age of Mythology use XS is different both from other XS systems and also from previous Age games. A scenario designer creates a series of simplified condition and effect clauses in the scenario editor, using a GUI. That data is stored in the scenario in a custom data format, and then the game dynamically generates XS to execute and runtime. This lets the trigger system utilize all the functionality and stability of XS, without requiring the burden of the full XS syntax in the more limited domain of triggers. It also allows the triggers to be limited in terms of how frequently they are executed, just as AI rules are.

It was ultimately very useful to have so many different systems using XS. Once the basics of XS were established, it provided a stable framework for easily generating a lot of content. The ability to leverage common utility systems like the integrated debugger made all of the XS systems instantly more powerful. It also allowed users of one system, who had already learned XS, to be able to more readily use the other systems.

We also paid a price for using XS in so many different ways. If we had just focused on the AI as originally designed, we doubtless could have cut corners or optimized the features and performance of the system for just that role. Ultimately we could justify spending more time on XS because of its key multipurpose role than we would have been able to had it just been for AI, and this probably led to a more powerful total system. Because it was so heavily used, there was also a higher psychological barrier to making significant changes.

Aside from the simple cost of developing our own custom scripting language and all the associated tools, one of the largest problems encountered on XS was with maintaining the integrity of the data. It was easy for the function signature of an XS call to change from any number of programmers, and then catching all the cases in every script could be difficult. Because XS did not have a standalone compilation mode, there was no way to do a “batch” compilation of all the XS files to find broken files. While normally an XS script was recompiled at run time to avoid problems with constants or database indices changing, this was a problem that had to be faced when trying to preserve XS state in a savegame. Understanding the importance of the standalone compiler earlier and adjusting the architecture to make that possible would have been a better approach to solving these problems.

Database

All of the actual data that defines the units, buildings, techs, and so forth for the game is stored in the game database. In addition to simple data fields like hit points, movement speed, and the like, the database could represent more complex data like defining the abilities of units, and effects and prerequisites of the technology tree. In essence, this data forms the entire definition of what the game itself is. Design had the capability to entirely rewrite the actual core game mechanics of the game by making changes to the database.

The data itself was kept in tables running on a SQL server, and accessed through a custom built MFC app using OLEDB. Using well established technology like SQL let us not have to worry at all about the integrity of the data and allowed us a great deal of data mining and filtering capabilities for free. It also allowed the database to be actively edited by multiple users as once, which was very convenient. Our actual SQL server ran using SQL Server 2000 on a fast dual-processor machine, which was probably overkill.

Not only did the database enable a lot of rapid changes in the data of the game, the database app itself was data driven. That is, the schemas that configured all the tables and such exposed through the MFC app were all generated by SQL tables themselves. When new data needed to be added to the DB app, it could be done without a new executable or even having to get people to exit out of the database!

Most of the actual text strings in the game (like names, help, etc.) were entered directly into the database MFC app. Because this information existed in application data, and not a separate data file, we could compile the strings into a string resource for localization without the content creators having to do anything at all.

The game itself did not access the SQL database, but rather the data was exported from the database application into a handful of XML files. Those XML files were then processed by the game to populate the actual in-game unit and tech databases. This made it very easy to spot check values being used in the game by manual inspection, and also allowed programmers to test local data by directly editing the XML files.

A lot of programmer time was invested in creating the custom built interface for editing the database tables. Having that front end proved to be an essential part of the design process for the database. The power of that tool gave the designers a great deal of freedom in rapidly making changes. SQL features

could be easily leveraged, but presented in a format that was well suited to easy adoption by the design team.

Like several other of the data formats migrated to XML, the database's export into XML was a great success. In addition to the process improvements mentioned above, it was useful for debugging and allowed us to do manual backup or editing for demos without having to pollute the real SQL tables. Because all of the programmers were very familiar with all the tools for reading the XML data, it was very easy to get new things added to the database and into the game.

The database export process did have a few problems, however. There were continually issues with tracking down integration problems with the code changing while the database had not been re-exported, or vice versa. A lot of code also had to be written to make the game robust when changes were happening in the database. For example, removing a tech from the game would cause the entirety of the tech IDs saved off in scenarios or savegames to become invalid, and so a dynamic remapping system had to be created.

Having the database centralized for so much important game data was very useful in maximizing the time invested on those tools. Unfortunately, it also increased the traffic of people in those files by a great deal, including a lot of people who hadn't been properly trained in how to use it. It only took a few episodes of nervous-looking programmers coming in and saying "uh, there's an undo button on this, right?" to demonstrate the importance of data recovery.

Using SQL as the core basis of the system also had some limitations that came from the underlying SQL tools that we were using. For example, the original design for the database called for the ability to save off "local copies" using the actual DB app directly, and this proved to be much harder than expected due to lower-level limitations in the system. However, none of those problems proved too troubling, and other mechanisms like the XML export provided easy workarounds.

Animation Control

The animation control system in Age of Mythology is responsible for resolving a unit's game state into an actual 3d mesh animation. Each unit in the game had a corresponding animation control file, which had entries for each possible kind of animation that the unit could play, and what the corresponding mesh filename was. Pretty much all 3d art in the game was channeled through this system, so it needed to be robust enough to handle many different cases, as well as having to be used directly by artists to hook up their own art.

In addition to just specifying simple animations like attacking or the like, the system also supported a definition system for using a system of "animation

selection logics" to further make decisions about which art to display. So a given unit might have a "tech logic" clause in their animation control file to allow different 3d models when different techs were researched. A given animation could also be set up to have a number of different variations so that it would choose among one at random. It could also be set up to allow parallel sets of animations within one file, as might be used for both male and female villagers for example.

Each animation clause could also be associated with various "tags", that specified callbacks into the game engine at a certain percent completion of the animation. For example, an archer's attack animation would have a tag for where the arrow should be created, and a tag for where additional sounds might be played.

The animation control files themselves used a custom text file format that allowed hierarchical specification of the logics, variations, and meshes to use. Almost all of the editing for these files was done with a plain text browser, though halfway through the project an in game tool was finally created for at least viewing the structure of an animation control file, and edit the tags. While the artists were the primary users of the system, every other department needed to edit them on occasion, so there was a lot of traffic on the files.

Use of the animation control files definitely helped the process at a high level. New units could go into the game, complete with fairly complex animation behaviors, without any programming involvement. The fact that all the art bugs could be handled autonomously within the art department was a great benefit to the process. It was also very easy for programmers to create new logics, and thus expose powerful new game behaviors to the artists.

However, this system, much more than almost all the other data driven systems in the engine, became a nightmare of complexity and difficulty. In theory, a lot could be done with no code or programmer time. In practice, the programmers had to be frequently involved in editing and debugging the animation control files just because of their sheer size and difficulty. By the later stages of the project, these files became the very definition of a Rube Goldberg problem, working but only through massive complexity. In fact, on several occasions decisions were made about other game features largely in order to avoid having to edit (and thus probably break) all the animation files in the game. The problem was further compounded by the fact that only a few people were capable of really using the animation control files, so that created a major bottleneck in the process both in creating new art files and in fixing existing problems.

Better tools would have helped the problem considerably. Once the in-game animation viewer went into the game, it at least helped to diagnose the problem. It was no substitute for what was really needed however: a tool for actually creating and maintaining the content of the animation control files. Not finding

the time to really create robust tools for the animation system was probably one of the worst tools related errors made on the project. Even if we stayed with a direct text editing format, converting over to XML would have at least provided a handful of external tools and eliminated the many syntax errors that continually plagued the system.

A much less critical secondary problem with the flexibility of the animation control system was related to performance. The engine spent a lot of time navigating each individual unit's animation tree, and this could add up to a real problem. Because the animation information was also used to determine simulation data like the bounding box of the unit, this tree navigation had to happen even for units that were not onscreen. Programmer time had to be spent to break that dependency and optimize the tree traversal of the animation system.

Lessons Learned

The first and foremost lesson we learned creating the various data driven systems on Age of Mythology, was that investment in tools is critical, and can easily pay for itself. Having a complex system without the proper tools to handle can completely undermine the utility of the system itself. It is unrealistic to expect non-technical people to suddenly develop the ability to grapple with complex programmer-oriented systems without a suitable set of tools. By creating those tools early enough, you can leverage them across the entire project, thus saving time that can be spent on other critical systems. Thus the tools earn a kind of "compound interest" in time, making it not only important to spend the proper time to make a useful time, but spending that time as early as possible.

On Age of Mythology we learned that lesson with both positive and negative experiences. When the proper time was spent on powerful tools like the database, it helped the adoption of that system vastly and was clearly a critical important investment in time. The animation system was continually hampered by a lack of tools, and both that and the UI layout system had severe problems until at least minimal tools were brought online. In both cases spending that time, or even more time, earlier, would have been a great investment in the project. While sometimes short term schedule pressures on any project can require a sacrifice of the long term for the short term, it is clear that the long term gains are very real when it comes to investment in tools.

Another useful experience from Age of Mythology was our success with XML. The low level XML tools were integrated into the game with very minimal effort, and having a consistency in file formats and, more importantly, data reading code, paid off throughout the entire project. The actual XML parsing was all encapsulated through an internal interface, so that we could keep a consistency of interface throughout the game without having to expose any of the arcana of XML. Common structure, external tool use, consistency in parsing, the ability to use external tools, and general robustness all were tangible advantages that

gave us real benefit on the project. While we had some initial resistance to adopting XML at first, especially among designers editing it “raw”, that feeling passed with time and eventually allowed us to leverage external XML editing tools to boost our tool development time without requiring any additional programming resources.

XML was used successfully primarily in the UI layout system and in the files generated by the database, but was also used in many smaller data files for systems like the god powers, in-game help, and many others. The few systems that we did not convert to XML, mainly due to a large body of “legacy” data, were a continual thorn in our side. The animation control files are the best and most direct example of this. For future game development, there is a very strong argument for adopting XML as fully as possible. Further information on XML can be found at http://www.xml.org.

Lastly, we had very mixed experiences with the kind of documentation needed to support a good data driven system. Obviously, having accurate and useful documentation is always a good thing for the users of your system, but wanting that kind of documentation and having it are two very different things. The only time we had real, long term, success with documentation was when it was either integrated directly into the system, or enforced by some external convention. Standalone documents describing the functionality of the system have their place, but short of a dedicated and consistent effort across the board, will inevitably fall into disrepair. Of course, even old documents can be useful compared to none at all, but finding a way to integrate the reference directly into your engine is by far the best way to keep it useful in practice.

In Age of Mythology, the only real successes we had for documentation were in the config and console systems, where the help could be accessed directly in game. More importantly, in those systems, the help information was embedded directly in the code, so that as more content was added it was continually updated. All of our experience in creating standalone reference documentation for systems like the UI layout and XS eventually fell into disrepair under the real pressures of game development.

In Closing…

While building a data driven system can be a lot of additional work, it is a great investment in the robustness, power and functionality of your game engine. More importantly, it can vastly improve the overall data flow process of game development, empowering the designers and artists to focus on their content, and keeping the programmers out of loop as much as possible. While we have had our share of rough spots, that philosophy has been heavily embraced in the creation of Age of Mythology, and should pay off great dividends for years to come.