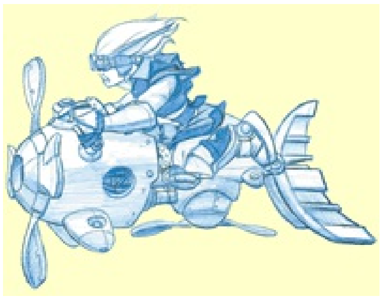
Postmortem: Naughty Dog's *Jak and Daxter: the Precursor Legacy*

By stephen white



By the end of 1998, Naughty Dog had finished the third game in the extremely successful *Crash Bandicoot* series, and the fourth game, *Crash Team Racing*, was in development for a 1999 year-end holiday release. And though Sony was closely guarding the details of the eagerly awaited Playstation 2, rumors - and our own speculations - convinced us that the system would have powerful processing and polygonal capabilities, and we knew that we'd have to think on a very grand scale.

Because of the success of our *Crash Bandicoot* games (over 22 million copies sold), there was a strong temptation to follow the same tried-and-true formula of the past: create a linear adventure with individually loaded levels, minimal story, and not much in the way of character development. With more than a little trepidation, we decided instead to say good-bye to the bandicoot and embark on developing an epic adventure we hoped would be worthy of the expectations of the next generation of hardware.

For *Jak & Daxter*, one of our earliest desires was to immerse the player in a single, highly detailed world, as opposed to the discrete levels of *Crash Bandicoot*. We still wanted to have the concept of levels, but we wanted them to be seamlessly connected together, with nonobvious boundaries and no load times between them. We wanted highly detailed landscapes, yet we also wanted grand vistas where the player could see great distances, including other surrounding levels. We hoped the player would be able to see a landmark far off in the distance, even in another level, and then travel seamlessly to that landmark.

It was important to us that Jak's world make cohesive sense. An engaging story should tie the game together and allow for character development, but not distract from the action of the game. The world should be populated with highly animated characters that would give Jak tasks to complete, provide hints, reveal story elements, and add humor to the game. We also wanted entertaining puzzles and enemies that would surpass anything that we had done before.
To achieve these and many other difficult tasks required three years of exhausting work, including two years of full production. We encountered more than a few major bumps in the road, and there were times when the project seemed like an insurmountable uphill battle, but we managed to create a game that we are quite proud of, and we learned several important lessons along the way.

What Went Right

1. Scheduling. Perhaps Naughty Dog's most important achievement is making large-scale games and shipping them on time, with at most a small amount of slip. This is an almost unheard of combination in the industry, and although there is a certain amount of luck involved, there are valid reasons to explain how Naughty Dog has managed to achieve this time and again.

Experience will tell you it's impossible to predict the precise details of what will be worked on more than a month or two in advance of doing it, yet many companies fall into the trap of trying to maintain a highly detailed schedule that tries to look too far into the future. What can't be effectively worked into these rigid schedules is time lost to debugging, design changes, over-optimism, illness, meetings, new ideas, and myriad other unpredictable surprises.

At Naughty Dog, we prefer a much more flexible, macro-level scheduling scheme, with milestone accomplishments to be achieved by certain dates. The schedule only becomes detailed for tasks that will be tackled in the near future. For example, a certain level will be scheduled to have its background modeled by a certain date. If the milestone is missed, then the team makes an analysis as to why the milestone wasn't achieved and changes plans accordingly: the background may be reduced in size, a future task of that artist may be given to another artist to free up more time, the artist may receive guidance on how to model more productively, or some future task may be eliminated.

Color Key for Forbidden Jungle. The game required two full-time conceptual artists for nearly two years of production.

In the case of *Jak & Daxter*, we used the knowledge we'd gained from creating the Crash Bandicoot games to help estimate how long it should take to model a level. As we modeled a few levels, however, we soon realized that our original estimates were far too short, and so we took appropriate actions. If we had attempted to maintain a long-term, rigidly detailed schedule, we would have spent a lot of time trying to update something that was highly inaccurate. Beyond this being a waste of time, the constant rescheduling could have had a demoralizing effect on the team.

2. Effective localization techniques. We knew from the start that we were going to sell Jak & Daxter into many territories around the world, so we knew we would face many localization issues, such as PAL-versus-NTSC, translations, and audio in multiple languages. Careful structuring of our game code and data allowed us to localize to a particular territory by swapping a few data files. This meant we only had to debug one executable and that we had concurrent development of all localized versions of the game.

All of our animation playback code was written so that it could automatically step animations at a rate of 1.2 (60fps/50fps) when playing in PAL. We also used a standardized number of units per second so that we could relate the amount of time elapsed in a game frame to our measure of units per second. Once everything was nice and consistent, then timing-related code no longer had to be concerned with the differences between PAL and NTSC.

Physics calculations were another issue. If a ball's motion while being dropped is computed by adding a gravitational force to the ball's velocity every frame, then after one second the ball's velocity has been accelerated by gravity 60 times in NTSC but only 50 times in PAL. This discrepancy was big enough to become problematic between the two modes. To correct this problem, we made sure that all of our physics computations were done using seconds, and then we converted the velocity-per-second into velocity-per-game-frame before adding the velocity to the translation.

3. Seamless world, grand vistas, and no load times. We knew very early on in the development of *Jak & Daxter* that we wanted to immerse the player within one large expansive world. We didn't want to stall the game with loads between the various areas of that world.

Jak & Daxter's designers had to overcome many obstacles to achieve our open environments. They had to lay out the levels of the world carefully so that levels could be moved in and out of memory without stalling gameplay or causing ugly visual popping. They also had to create challenges that would engage the player and maintain the player's interest, even though the player could roam freely around the world. And they had to tune the challenges so that the difficulty ramped up appropriately, without giving players the impression that they were being overly directed.

The programmers had to create tools to process interconnected levels containing millions of polygons and create the fast game code that could render the highly detailed world. We developed several complex level-of-detail (LOD) schemes, with different schemes used for different types of things (creatures versus background), and different schemes used at different distances, such as simplified models used to represent faraway backgrounds, and flats used to represent distant geometry. At the heart of our LOD system was our proprietary mesh tessellation/reduction scheme, which we originally developed for Crash Team Racing and radically enhanced for *Jak & Daxter*.

The artists had the burden of generating the enormous amount of content for these environments. Their task was complicated by the very specialized construction rules they had to follow to support our various renderers. Support tools and plug-ins were created to help the artists, but we relied on the art staff to overcome many difficulties.

4. Camera control. From the initial stages of*Jak & Daxter*, we looked at the various camera schemes used in other games and came to the depressing conclusion that all existing camera schemes had serious issues. We suspected that making a well-behaved camera might be an unsolvable 3D problem: How could one possibly create a camera that would maneuver through a complex 3D world while behaving both unobtrusively and intelligently?

Only fools would believe that all problems have a solution, so, like idiots, we decided to give it a try. The resulting camera behaved extremely well, and although it had its limitations, it proved the problem does indeed have a solution. Jak can jump through trees and bushes, duck under archways, run between scaffolding, scale down cliffs, and hide behind rocks, all with the camera unobtrusively keeping the action in view.

We wanted the player to be able to control the camera, but we did not want to force the player to do so. Players can use the second joystick to maneuver the camera(rotating the camera or moving it closer to or farther from Jak), but we were concerned that some people may not want to manipulate the camera, and others, such as children, may not have the required sophistication or coordination. Therefore, we worked very hard at making the camera do a reasonable job of showing players what they needed to see in order to complete the various challenges. We accomplished this through a combination of camera volumes with specially tuned camera parameters and specialized camera modes for difficult situations. Also, creatures could send messages to the camera in order to help the camera better show the action.

This may sound funny, but an important feature of the camera was that it didn't make people sick. This has been a serious problem that has plagued cameras in other games. We spent a bit of time analyzing why people got sick, and we tuned the camera so that it reduced the rotational and extraneous movement that contributed to the problem.

Perhaps the greatest success of the camera is that everyone seems to like it. We consider that a major accomplishment, given the difficulty of the task of creating it.

5. GOAL rules! Practically all of the run-time code (approximately half a million lines of source code) was written in GOAL (Game Object Assembly Lisp), Naughty Dog's own internally developed language, which was based on the Lisp programming language. Before you dismiss us as crazy, consider the many advantages of having a custom compiler.

Lisp has a very consistent, small set of syntactic rules involving the construction and evaluation of lists. Lists that represent code are executed by evaluating the items that are in the list; if the head of the list is a function (or some other action), you could think of the other items in the list as being the parameters to that function. This simplicity of the Lisp syntax makes it trivial to create powerful macros that would be difficult or impossible to implement using C++.

Writing macros, however, is not enough justification for writing a compiler; there were features we felt we couldn't achieve without a custom compiler. GOAL code, for example, can be executed at a listener prompt while the game is running. Not only can numbers be viewed and tweaked, code itself can be compiled and downloaded without interrupting or restarting the game. This allowed the rapid tuning and debugging, since the effects of modifying functions and data structures could be viewed instantaneously.

We wanted creatures to use nonpreemptive cooperative multi-tasking, a fancy way of saying that we wanted a creature to be able to execute code for a while, then "suspend" and allow other code to execute. The advantage of implementing the multi-tasking scheme using our own language was that suspend instructions could be inserted within a creature's code, and state could be automatically preserved around the suspend. Consider the following small snippet of GOAL code:

```
(dotimes (ii (num-frames idle))
(set! frame-num ii)
(suspend)
)
```

This code has been simplified to make a point, so pretend that it uses a counter called ii to loop over the number of frames in an animation called idle. Each time through the loop the animation frame is set to the value of ii, and the code is suspended. Note that the value of ii (as well as any other local variables) is automatically preserved across the suspend. In practice, the preceding code would have been encapsulated into a macro such as:

```
(play-anim idle
;; Put code executed for each time..
;; through the loop here.
)
```

There are other major compiler advantages: a unified set of assembly op-codes consistent across all five processors of the Playstation 2, register coloring when writing assembly code, and the ability to intermix assembly instructions seamlessly with higher-level code. Outer loops could be written as "slower" higher-level code, while inner loops could be optimized assembly.

What Went Wrong

1. GOAL sucks! While it's true that GOAL gave us many advantages, GOAL caused us a lot of grief. A single programmer (who could easily be one of the top ten Lisp programmers in the world) wrote GOAL. While he called his Lisp techniques and programming practices "revolutionary," others referred to them as "code encryption," since only he could understand them. Because of this, all of the support, bug fixes, feature enhancements, and optimizations had to come from one person, creating quite a bottleneck. Also, it took over a year to develop the compiler, during which time the other programmers had to make do with missing features, odd quirks, and numerous bugs.

Eventually GOAL became much more robust, but even now C++ has some advantages over GOAL, such as destructors, better constructors, and the ease of declaring inline methods.
A major difficulty was that we worked in such isolation from the rest of the world. We gave up third-party development tools such as profilers and debuggers, and we gave up existing libraries, including code previously developed internally. Compared to the thousands of programmers with many years of C++ experience, there are relatively few programmers with Lisp experience, and no programmers (outside of Naughty Dog) with GOAL experience, making hiring more difficult.

GOAL's ability both to execute code at the listener and to replace existing code in the game at run time introduced the problem of memory usage, and more specifically, garbage collection. As new code was compiled, older code (and other memory used by the compiler) was orphaned, eventually causing the PC to run low on free memory. A slow garbage collection process would automatically occur when available memory became sufficiently low, and the compiler would be unresponsive until the process had completed, sometimes taking as long as 15 minutes.

2. Gameplay programming. Because we were so busy creating the technology for our seamless world, we didn't have time to work on gameplay code until fairly late in the project. The situation caused no end of frustration to the designers, who were forced to design levels and creatures without being able to test whether what they were doing was going to be fun and play well. Eventually programmers were moved off of technology tasks and onto gameplay tasks, allowing the designers to play the game and make changes as appropriate. But without our designers' experience, diligence, and forethought, the results could have been a disaster.

3. Audio. We were plagued with audio-related problems from the start. Our first indication that things might not be going quite right was when our sound programmer quit and moved to Australia. Quickly hiring another sound programmer would have been the correct decision. We tried several other schemes, however, made some poor choices, and had quite a bit of bad luck. We didn't recognize until fairly late in development what a monumental task audio was going to be for this project. Not only did *Jak & Daxter* contain original music scores, creature and gadget noises, ambient sounds, and animated elements, but there are also over 45 minutes of story sequences, each containing Foley effects and speech recorded in six different languages.

Our audio issues could be broken up into four categories: sound effects, spooled Foley, music, and localized dialogue. Due to the large number of sound effects in the game, implementing sound effects became a maintenance nightmare. No single sound effect was particularly difficult or time-consuming; however, creating all of the sound effects and keeping them all balanced and working was a constant struggle. We needed to have more time dedicated to this problem, and we needed better tool support.

We used spooled Foley for lengthy sound effects, which wouldn't fit well in sound RAM. Spooling the audio had many advantages, but we developed the technology too late in the project and had difficulty using it due to synchronization issues.

Our music, although expertly composed, lacked the direction and attention to detail that we had achieved with the *Crash Bandicoot* games. In previous games, we had a person who was responsible for the direction of the music. Un-fortunately, no one performed that same role during *Jak & Daxter*.

Dialogue is a difficult problem in general due to the complexity of writing, recording, editing, creating Foley, and managing all of the audio files, but our localization issues made it especially challenging. Many problems were difficult to discover because of our lack of knowledge of the various languages, and we should have had more redundant testing of the audio files by people who were fluent in the specific languages.

4. Lengthy processing times. One of our greatest frustrations and loss of productivity came from our slow turnaround time in making a change to a level or animation and seeing that change in the actual game.

Since all of our tools (including the GOAL compiler) ran across our network, we ran into severe network bandwidth issues. Making better use of local hard drives would have been a smarter approach. In addition, we found extreme network slowdown issues related to reading file time/date stamps, and some tools took several minutes just to determine that nothing needed to be rebuilt. When we compiled some of our tools under Linux, we noticed dramatic improvements in network performance, and we are planning on using Linux more extensively in our next project.
We implemented the processing of the lengthy story-sequence animations as a hack of the system used to process the far simpler creature animations. Unfor-tunately, this bad system caused lengthy processing times, time-consuming debugging, and a lot of confusion. If we had initially hidden the processing complexity behind better tools, we would have saved quite a bit of time.

We used level-configuration scripts to set actor parameters and other level-specific data. The script processing was done at an early stage in our tools pipeline, however, so minor data changes took several minutes to process. We learned that tunable data should instead be processed as close as possible to the end of the tools pipeline.

5. Artist tools. We created many tools while developing*Jak & Daxter*, but many of our tools were difficult to use, and many tools were needed but never written. We often didn't know exactly what we needed until after several revisions of our technology. In addition, we didn't spend a lot of time polishing our tools, since that time would have been wasted if the underlying technology changed.

Regrettably, we did not have time to program tools that were badly needed by the artists, which resulted in a difficult and confusing environment for the artists and caused many productivity issues. Since programming created a bottleneck during game production, the added burden given to the artists was considered necessary, though no less distasteful.

We lacked many visualization tools that would have greatly improved the artists' ability to find and fix problems. For example, the main method artists used to examine collision was a debugging mode that colorized a small section of collision geometry immediately surrounding Jak. A far better solution would have been to create a renderer to display the entire collision of a level.

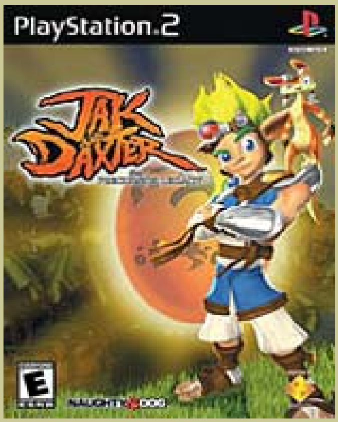
We created plug-ins that were used within the 3D modeling package; however, for flexibility's sake most of the plug-ins operated by taking command parameters and outputting results as text: not a good interface for artists. Eventually, one of our multi-talented artists created menus and other visualization aids that significantly improved productivity.

Many of our tools were script based, which made the tools extremely flexible and adaptable; however, the scripts were often difficult for the artists to understand and use. We are replacing many of these scripts with easier-to-use GUIs for our next project.

The Legacy

Creating *Jak & Daxter* was a monumental effort by many hardworking, talented people. In retrospect, we were probably pretty foolish to take on as many challenges as we did, and we learned some painful lessons along the way. But we also did many things right, and we successfully achieved our main goals. At Naughty Dog, there is a strong devotion to quality, which at times can border on the chaotic, but we try to learn from both our success and our failures in order to improve our processes and create a better game. The things that we learned from Jak & Daxter have made us a stronger company, and we look forward to learning from our past as we prepare for the new challenges ahead.

Game Data



Jak & Daxter: the Precursor Legacy

Publisher: Sony Computer Entertainment

Number of full-time developers: 35

Length of development: 1 year of initial development, plus 2 years of full production

Release date: December, 2001

Platform: Playstation 2

Development software used: Allegro, Common Lisp, Visual C++, Maya, Photoshop, X Emacs, Visual Slick Edit, tcsh, Exceed, CVS

Return to the full version of this article