GAMASUTRA
The Art & Business of Making Games

PRINT

Postmortem: Freeverse's Marathon 2: Durandal

By Mark Levin

Freeverse, primarily a Mac developer, first came to Microsoft's attention thanks to the camera technology developed by our partner studio, UK-based Strange Flavour. We established our own Xbox 360 group during the development of their *TotemBall* and *Spyglass* board games, and were looking for something to apply it to when an extremely serendipitous conversation -- in, of all things, *World of Warcraft* -- gave us the opportunity to do the Xbox Live Arcade port of *Marathon 2*: *Durandal*.

The project was one that was oddly resonant with the history of both the company and its people. In the golden days of the mid-90's, there was only one word worth speaking about games on the Macintosh, and it was "*Marathon"*. Practically everyone who worked extensively with Macs or in an environment with a good number of them on a network knew and loved the game -- a category which includes most Freeverse personnel. When we landed it, we were excited not just for the chance to revisit an old favorite, and not just to port it, but for the chance to truly bring it into the modern era.

Our major goal for the project was to go beyond the standard exemplified by the majority of classic games that had made it to XBLA in the past. Emulator-based ports are all well and good, but they tend to leave the perks of their new home as second-class citizens.

Leaderboards and achievements may feel tacked-on or irrelevant, fullscreen graphics (or even older vertical-tube arcade games) look clumsy when stretched or padded on widescreen displays, and control schemes sometimes mesh badly with the 360's controller. And, as longtime fans of the game, we were sure we could come up with some tweaks and extras that would add value to the game beyond simple nostalgia.

What Went Right

Going beyond a port. Without an emulator, huge chunks of the game had to be discarded. The original rendering code of the game was very well suited to its original platform, but was entirely unusable on the Xbox 360. MacOS's proprietary graphics API, QuickDraw, was used extensively throughout; some of its features, such as drawing 2D lines, were absent from the 360 entirely. Screen-space arithmetic was done using fixed-point integer math, a crucial optimization a decade ago but a source of needless complexity today.

Texture mapping was done using blocks of PowerPC and 68K assembler -- no one on the team was able to directly read them. The entire renderer had to be rewritten in DirectX. As a result, the 360 version of *Marathon 2* outputs a native 720p image at 60 frames per second, rather than the 480p30 of the original. And unlike most other classic ports, it uses texture filtering, true 32-bit rendering, and every pixel of a

widescreen display.

The "HD" mode of the graphics, which in other games commonly refers to a post-processing filter, is a completely new set of images. All of the nearly three thousand sprites in the game were redrawn in true 32-bit color, allowing us to put our own spin on the look of the game. Enemy sub-variants are no longer simple palette swaps, but have significant differences in their appearance. Muzzle flashes are now transparent, and the weapons have a modernized look that's still faithful to the original game. As a result, the game's content holds up much better at 720p; the dithering visible in the 8-bit sprites is eliminated.

The heads-up display of the original game was simply huge. It consumed nearly a third of the screen vertically, and displayed many different screens of tightly packed text. This was unsuited to the Xbox 360 version in several ways, not the least of which being that when moved to a 16:9 display, there was practically no space left for the world view!

We redesigned the HUD by splitting it into several pieces, preserving the rough layout of the information it presented but revealing far more of the world behind it. The text sections were converted to icons and numbers large enough to be easily readable on a television. The improved HUD takes up far less of the screen than the original while conveying the same information.

A team with genuine love for the project. The largest problem we encountered while updating the game engine was how to preserve the feel and flow of the gameplay, but since everyone on the team was already intimately familiar with that gameplay it was easy to get a handle on issues. The change from 30 to 60 frames per second necessitated a change from 30 to 60 world ticks per second, because there was insufficient separation of updates and renders within the engine.

In most cases this was trivial to compensate for, but some areas of the game had been carefully tweaked with fudge values for the original game's 30 fixed-point evaluations per second, and had to be re-tweaked in the new engine. Our experience helped get the game's feel as close as possible to the original without time-consuming tweaking and test iterations.

Conformance with the original game was always the highest priority when modifying the engine. Aleph One, the open-source version of the *Marathon 2* engine, was invaluable to the project as a reference implementation that could be examined in a debugger (the original *Marathon 2* had been written with CodeWarrior, but so long ago that the project file was no longer readable by any current version).

This was invaluable for ensuring the correct behavior of the monster AI, which picked up a number of bizarre quirks during the port. Monsters would rapidly twitch back and forth in places where the correct response would be to simply reverse direction. By comparisons with Aleph in the debugger, this was eventually traced to an error in renaming conflicting variables.

A pre-existing familiarity with the game's levels came in handy when it was time to test. Knowing the location of all the secret areas, which tend to have tighter tolerances for movement and timing, made for much more efficient testing, as many problems could be caught by a coder during a cursory examination. *Marathon* allows for "grenade hopping", a sort of primitive form of rocket jumping, and many secrets require its precise use to clear jumps by inches; all of these had to still be reachable on the Xbox version.

Another benefit was that a large number of reports from testers could be dismissed immediately as conforming to the original game, even if they were unusual by modern standards, such as the bizarre behavior of monsters pelted with a steady stream of bullets- they'd smash into the wall, slide up until they hit the ceiling, and then hang their for several seconds before falling to the floor.

Our relationship with Microsoft. The Xbox Live Arcade team was an invaluable part of the project. In addition to being a ready conduit for technical questions, they provided us with many resources during the project. Implementing text flow in a way that supported every locale we wanted to target is extremely nontrivial, but we were given an internal library that provided it. When optimizing the save game screen, I was able to send off a profiler log for expert analysis. This direct assistance made development on the Xbox 360 a much smoother experience than it could have been.

If that wasn't enough, our account manager was available on MSN Messenger for most of the project. Questions about the certification requirements or some aspect of the 360 platform could be answered very quickly, and being able to simply converse with veteran Microsoft personnel gave us insight into the platform's nature and intended functioning. Uncertainty stemming from any of the 360's requirements never held up the project for long, and with the Arcade team's guidance we were able to bring the project in on time and without having to crunch for more than a handful of weeks. And only one person ended up sleeping in the office.

A flexible and cross-platform toolchain. We decided early on in development that we wanted to keep the original game data intact wherever feasible. The theory was that it would remove sources of uncertainty in the conversion process -- existing loader code would provide a completely reliable method of establishing assets in RAM, and the only potential sources of bugs would be later on in the gameplay code. This theory did not survive being put into practice, but when the dust settled we were much better off after all.

The problem was the high-resolution sprite loader. We had already abandoned the original sprite collection format in favor of 32-bit PNGs, but the tens of thousands of seeks the game performed when launched were unacceptable. Collecting the assets into a small number of package files meant that we were no longer strictly loading the original data, but it also allowed us to shave over ten seconds off the load time, a very fair trade. Later in development, the package file system also allowed us to integrate compression, which turned out to be essential to reaching XBLA's 150-megabyte file size cap.

The package files necessitated the creation of a tool to assemble them, but other tools were also created over the course of the project. *Marathon* originally stored its sprites and animation data in the same place -- a single filed named "Shapes". Once the decision was made to move sprites to individual PNG files, the copies of the images within the shapes file became redundant, a waste of about 10 megabytes of space. A tool had to be created to gut the Shapes file, leaving only the animations. It was at this point one of the unexpected advantages of being a Mac-centric studio was noted -- the (PowerPC) Macintosh and the Xbox 360 are both big-endian.

Load times were still unacceptable, so one final optimization was performed on the package file format. It was found that the package file was being searched with simple string comparisons (each chunk within it was referred to by what its path had been before packing), and the work done to iterate over this table was adding up. The packaging tool was changed to hash the paths and store the resulting table in the package instead of a simple mapping of path strings to locations. Integrating this into the resource loading system was easy thanks to the reliance on packages, and it shaved around 10 seconds off the load with virtually no cost in reworking other areas of the game.

Leaderboard-linked Achievements. There are two leaderboards in*M2:D* that record the time at which a player earned an Achievement -- one for completing the single-player campaign while obeying some restrictions (the "vidmaster" rules), another for a "viral" achievement awarded for kills in network games. This was proven to be a good idea when the first few entries on the board were deluged with Xbox Live messages, most asking for assistance in earning their own place on the board. The leaderboards practically became an alternative friends list, as existing vidmasters hosted cooperative games to induct new ones.

The standard Xbox 360 Achievement system allows players to compare their Achievements, but only to one friend at a time. The idea of building a global list of Achievement earners added a new facet to the feature -- the chance to move faster than other players and to be higher on the list.

What Went Wrong

Failure to anticipate platform and API design requirements. Many of the required features of the Xbox 360 platform were implemented hastily after-the-fact only after their absence was discovered, rather than being integrated into the original design of the code. Guest support on Xbox Live is a good example of this. When a player joins a split-screen multiplayer game, he must be able to bring players who do not have personal Live accounts with him.

By the time we realized this feature was not working correctly, a very strong relationship had been established between player accounting in the network game UI, player representation in the game world, and player profiles on the Xbox. Supporting players who did not have matching profiles required a significant overhaul of that accounting system. An extra wrinkle came from the fact that profiles have privileges associated with them that may require a player with a profile to be treated as a guest anyway under certain circumstances -- in other words, game developers must themselves implement the difference between Xbox Live Silver and Gold accounts. The system used by the final game relies on the persistence of data in partially re-initialized structures, an unfortunate compromise that should have been avoided.

The performance requirements of the platform also had a huge influence on our design. A smooth, stable framerate is important to a good presentation, and to the proper functioning of the Xbox Guide, so no blocking could occur when dealing with files or the network. As the Xbox 360 uses polling for most asynchronous operations, this necessitated the use of state machines in many places. The most egregious example of this is the game saving and loading screen, which is running two separate state machines at the same time -- one to update the UI, and one to perform the file I/O.

The number of synchronization bugs and mishandled edge cases resulting from this was frightening. Another tricky spot was the loading screen. Most of the time, our engine uses a dedicated thread to manipulate the Direct3D device object, but the bitmaps that needed to become texture objects were being loaded by a different thread. The device had to be switched over to the loading thread and the load screen rendering had to be interleaved with the actual loads, then the device returned to the render thread. At other places in the engine, we needed to be able to show the progress screen but were not able to tick it "by hand" during a lengthy process, so the load screen class also had to be able to update and render itself without being ticked by another object.

Sometimes the requirements for certification diverged from the functionality provided by the system libraries. The mute feature, for example, must enforce muting in both directions -- when a player mutes another, neither the muter nor the mutee may hear the other. The catch is that the list of muted players is only available for local players; it is not automatically propagated. The mutee doesn't know he has been muted until the game implements a method of notifying him.

Once this was done, a bug was found (in our game code) by which a user could force another user to un-mute him by using his own mute button. The was so late in the project that rather than redesign the mute system additions, a "countermand" message was created so that different consoles could order each other to re-mute players when necessary, wasting network traffic and complicating the related code. And it’s still not perfect -- while it passed cert, we've heard several reports of players being unable to communicate when appropriate.

Flaws in the team. The personnel we assembled for this project had all been Mac and Windows developers for years, but among them there was little porting experience, even less experience with console development, and for most of us this was the largest single team we had ever been a part of. Some of the items listed in this postmortem may seem obvious or mundane to more experienced readers -- well, it was new to us at the time. We were expanding our own envelopes. In theory, the presence of console veterans "on call” elsewhere in the company and our partner studios would keep everyone on track.

For technical aspects of the project, this largely happened. Problems we encountered when trying to design for stability or puzzle out a particular quirk of the 360 never held us up for long. Process and design were another story. Communication between team members was infrequent, the master design document was never fully fleshed out, and many functional areas of the game were laid out by individuals working in a virtual vacuum.

The audio system was the work of a single coder, as was network session management, the menu interface, and user profile data. As these largely complete units began to be integrated, poor communication led to duplication of effort and an inability to keep up with changing spheres of responsibility. Many hours were lost when it was discovered that two programmers had been working on the same code -- optimizing the Shapes loader cost nearly an entire extra man-day for this reason.

Another problem caused by this deficiency was that due to different coding styles and lack of a comprehensive design, many areas of the code could only be efficiently debugged by the person who had created them. The audio system in particular involved multiple threads and extensive use of the Xbox 360’s audio library, which no one except the coder directly responsible for it was sufficiently familiar with.

As a result, it continued to produce bug reports caused by subtle race conditions throughout practically the entire project -- reports which most of the team couldn’t address. Even when singular code like this wasn’t producing bugs, it was causing coders to run into trouble and waste time wrestling with largely undocumented APIs and quirks when their work brought them into contact with another functional area.

Drop-in libraries are not a magic spell. For our network code, we hired an experienced programmer who provided us with his pre-existing network library. We expected that adding it to the project would take minimal effort. Unfortunately, this was not the case. Despite being a very flexible and powerful system, it had still been built with certain expectations in mind and made assumptions about the code it would interact with. It relied on constructors and destructors to perform much of its work, while our game used statically allocated arrays of objects to build the game world. The first time we attempted to invoke the network code while running the game, it simply exploded. Determining why this happened took several days, since the flaw was in such a fundamental area.

The eventual solution was to establish a system of proxy objects whose creation and deletion followed the rules the network library expected, and which manipulated the static arrays to control entities in the world. This was far from an ideal solution, as innumerable bugs appeared in the synchronization between the two objects, and it removed a major advantage of the static array design -- a completely predictable memory layout.

Overhauling very old code is difficult. *Marathon* was written in straight C, which does not lend itself to good software engineering (and that's not even considering the requirements of high-performance 3D graphics in 1994). Most interaction between different functional areas was accomplished by direct manipulation, not accessors. The life cycle of particular parts of the world was quite difficult to puzzle out in many cases. Quite often, the limitations of the original 68K Macintosh platform had been worked into the gameplay in subtle ways.

Multiplayer scoring was a particularly hairy section to overhaul. The original game had used a pure peer-to-peer network architecture that only transmitted raw user input -- usually passable in a LAN environment, unworkable on the internet. Since the game did not differentiate between remote and local players in any meaningful way, scorekeeping was implemented simply by giving each player a table of which opponents they had beaten or fallen to and having each player grant points to the appropriate enemy (through a direct write to his internal table, naturally) when dying.

When the model was changed to transmit absolute states over a client-server network, this had to be replaced with a message-passing system that informed remote players of their earned kills -- more accurately, it informed a remote player of his kill and then propagated his updated score to every other client in the game, to satisfy Xbox Live's requirements for verifying game coherence.

The sometimes strange organization of the engine led to one particularly thorny problem with the weapon-in-hand animations. For a long time after the functionality was ported, the feel of the weapons was slightly off. Guns would fire too quickly or not quickly enough, and animation glitches and timing problems were rampant, but there was no clear data corruption or logical error to be found. The animation of the weapons was all correctly loaded, as was the "physics model" file that defined the rest of their behavior, but the two data sets were inconsistent and had always been.

It was eventually discovered that there was a preprocessing function that synchronized the two that had been left out entirely -- its only invocation was in an obscure area of initialization that had been superseded by an Xbox 360-specific rewrite. The question of why one set of conflicting data wasn't chosen to be canonical and used in its original loaded form was never answered.

Fixed-point arithmetic was used throughout the original game, and it was sometimes reflected in the gameplay in unexpected ways. *Marathon* has two features that were fairly unique for its time: liquid volumes with variable height and currents that push the player around, and wall panels that can be used to recharge health. We received a bug for a level late in the game -- there was an underwater recharger that could not be used, since the liquid would push the player away from it. Why didn't the liquid push the player away in the original game?

The answer lay in the fact that the liquid current was now stored in a float rather than a fixed-point integer. The exact value of the liquid's

force was applied to the player at reduced strength (the full value was used to animate its flowing surface); a division by 4 for a float, a right shift of 2 bits for a fixed. Most of the time, these gave similar results, but what if the current flow was already less than 4? The fixed-point math drops it all the way to zero; the float leaves it at a very small positive value. The underwater recharger worked because of this inaccuracy; once it was manually restored the player could recharge again.

Was an updated port a good idea in the first place? Why do people play old games, especially old games they have never played before? Some do it for nostalgia, to reinforce their memories of when the game was brand new. Some do it to satisfy their curiosity about an experience that others feel is worth remembering even after countless other games were released. Some do it to witness the evolution of modern games, and see brand new concepts in their unrefined states. Fundamentally, the goal is the same regardless -- to go back and experience the game as it once was, to see for oneself the greatness that was once discovered in it without preconceptions or the relaxed standards of hindsight.

We had decided to go in a different direction. We felt the core of what made the game great -- the level design, the feel of working the game's controls and weapons, the story, deserved a stronger technological foundation. Some of the game's limitations added nothing to it and could be discarded, and additions could be made that enhanced the parts of the experience we felt were lacking. In hindsight, this was a fundamental mistake. Fidelity to the original should have been absolutely paramount, and it could have been done while playing to the 360's strengths all along. Our long experience with the game and its tightly-knit original community led us to fail to consider what a player approaching it for the very first time was expecting to find.

Practically every other port on Xbox Live, and older games running under emulators or virtual machines on PCs, can be made to look so similar to the original that it's practically impossible to distinguish screenshots of the two. Ours cannot. Texture filtering is always used; the blocky nearest-neighbor look of software texturing is gone. Vertical foreshortening always occurs in the truly 3D renderer. The new HUD, while unavoidable for reasons stated above, is nevertheless new. The port's 60 fps performance makes some effects carefully tweaked for half that rate look strange, such as the flamethrower or the rocket's contrail.

The high-quality mode is not simply a tweaked version of the standard, but a completely new look that warps the aesthetics of some areas in unexpected ways. These changes may be superficial to the gameplay, but they are pervasive and unavoidable and they make the game a unique experience to old and new players alike; an experience that can't decide whether it wants to be judged as a modern game weighed down by gameplay and rough edges a decade old, or a relic of the past that tries to deny its players the very reason they sought it out. Lost in the gap between contemporary expectations of polish and the implicit forgiveness in gaming archaeology, the experience is not exactly what we had intended for any one player.

Conclusion

Just because a project is a port of an older game does not imply that it will be less complex. Overhauling systems not suitable for use on a different platform can involve a great deal of work, and console platforms with certification processes may require the creation of large swaths of brand-new code and content. But while all this new work is performed, the core goal of the project must not be lost -- a port is a chance for an old game to have another chance at entertaining a new audience.

Project Stats

Developer: Freeverse
Publisher: Microsoft Game Studios
Platform: Xbox 360 (Xbox Live Arcade)
Release date: August 1, 2007
Development time: 10 months
Number of developers: 5
Hardware: Mac minis running Windows XP under Boot Camp
Development software: Visual Studio 2005, Adobe CS 2.0, Xcode
Technologies: DirectX
Lines of code: 42,000
Budget: $300,000

Return to the full version of this article