GAMASUTRA
The Art & Business of Making Games

PRINT

Postmortem: Activision's *Heavy Gear 2*

By Clancy Imislund

After a long series of successful titles such as *MechWarrior 2: Ghost Bear's Legacy* and *MechWarrior 2: Mercenaries*, Activision held a dominant position in the giant-robot genre. Due to the commercial success of this series, though, a tidal wave of similar products developed by worthy competitors began to flood the market. Fortunately, Activision found a new and exciting universe in Dream Pod 9's *Heavy Gear* pen-and-paper-based game, and in the fall of 1997, Activision Studios began production on the futuristic giant-robot simulator *Heavy Gear II*.

Heavy Gear II allows game players to suit up in a giant, high-octane, humanesque battle tank called a "Gear." The player is then required to outfit a wily band of AI Gears (called "squadmates") and arm them to the teeth. Their small but heavily armed reconnaissance force must infiltrate and overcome a rich variety of environments including swamps, icy wastes, angry red planets, and the weightless reaches of outer space. The action in *Heavy Gear II* is much faster than in other giant robot simulators, as Gears are substantially smaller than Mechs and much more fragile. This requires squads to rely primarily on stealth and cunning rather than brute force and wanton destruction. Players are forced to pick their battles wisely or face annihilation from the superior enemy legions that infest the various worlds they traverse. When pilots are tired of playing against computer opponents, they can pit themselves against human enemies in multiplayer mode with a number of game types to choose from including "steal the beacon," "king of the hill," and good old-fashioned deathmatch.

To create such an experience, Activision assembled a new team of its best and brightest in the areas of programming, art, design, and management. A very aggressive schedule was adopted targeting a fall 1998 release date (a 13-month development schedule). Our original staff consisted of a lead, graphics, AI and tools, multiplayer and shell programmer. We had a lead artist, two 3D modelers, a 3D animator, and two 2D artists creating texture maps for the models and terrain. We also had a lead designer who managed a group of three game designers, all of whom transformed the storyline from paper into a computer game. Our management group consisted of a director, a producer, and an associate producer, who controlled the schedule and the development and direction of the game and saw to the many needs of the other production staff.

We programmers were chartered to create a new game engine and I took on the role of AI, scripting system, and tools programmer. At the heart of this engine was a rock-solid memory management and leak-tracking class. Yes, that's right. Before anyone ever dreamed of fancy graphics or stunning game play, we had to deal with this mundane task. Every C++ class and structure used by the Darkside engine had roots within this base class. This architecture allowed to us to detect memory leaks and overwrites as soon as they appeared in a given day's build, which allowed us to address problems immediately rather than during a grueling cycle at the end of the project. I cannot stress the importance of this type of planning and execution enough for teams who want to craft a state-of-the-art game engine. Focusing on the reliability of the application will also greatly increase the immersion factor of any game that's created with it. After all, what destroys immersion more than a hard system crash? Hats off to our lead who took us down this path.

The decision was made to target the game only for machines outfitted with 3D-accelerated video cards. This issue was hotly disputed within our team for a while, but when our management group realized that in order to pull off realistic 3D graphics, software emulation was a dead deal. This had profound effects on the schedule, as it freed our artists and programmers from dealing with the time-consuming and tiresome work of generating alternate LODs for such a purpose. This decision also eliminated a huge chunk of our QA test plan, which could have pushed back the release date of the title significantly further. We had to choose between putting out the highest quality game we could and targeting a consumer market that barely existed at that point, or publishing a game that had a greater current market base and little or no novelty by the time it was released. I suppose it is something all of us developers must deal with in these days of rampant technological change.

Artist's concept of the "Asteroid Shipyards."

The AI system was another great challenge, as it would dictate much of the feel of the game. It would also be a key tool for our design team as they implemented the complex storyline. We decided at the high level to go with a pure "autopilot" approach, in which enemy and friendly AI alike were able to function intelligently without any script at all for individual units. We wanted to be able to put a unit into the world and have it go do whatever it thought it needed to do. Internal to this was a high-level strategic system, a team-based tactical system that employed a knowledge base, and a low-level, unit-specific order execution system. The goal was to take much of the burden off of the design staff, and give the game a consistent and distinctive feel across all missions.

Nonetheless, our scripting system was very effective and had some nice features such as an in-game debugger complete with break points, single-step execution, and variable watch windows. It was based on LEXX/YACC-based C grammar, which hooked into a powerful and easily extensible virtual machine that resided in the Darkside engine. Our scripting system was hooked in to our custom game API, which in turn was coupled to prototypes defined in a file called STDHG2.H, which ostensibly replaces STDIO.H (a file well known to all of you C programmers out there). Amazingly enough, STDHG2.H was used not only to compile the scripts, it was also used in the compilation of the Darkside engine itself. This convenient relationship between the scripting language and the engine source code, plus the fact that the C language is widely documented, easily justified our decision use a scripting language based on C. Scripts were used mainly to monitor mission progress and objectives, special behaviors (convoys and patrols, for instance), and interactive control of the action. Our goal was to keep these scripts as simple as possible. The autopilot handled all strategic, tactical, and low-level operations of unit behavior, yielding control only at the request of a script. The autopilot was the default AI handler in the engine. Scripts could override this system by posting event callback requests as shown in Listing 1.

Listing 1. Event Callback Requests

```
// A Script.C
void
Initialized()
{
// Tell AutoPilot that we want control
// back when the unit is killed or
// shot at
AutoBreak(HitPointsExhausted);
AutoBreak(ShotAt);

// Turn over control to the autopilot
// when this function returns
AutoPilot();


}

void
ShotAt()
{
RegisterString("Ha ha, ya missed");
// Resume auto mode
AutoPilot();
}
void
HitPointsExhausted()
{
RegisterString("Damn! I died!");
}
```

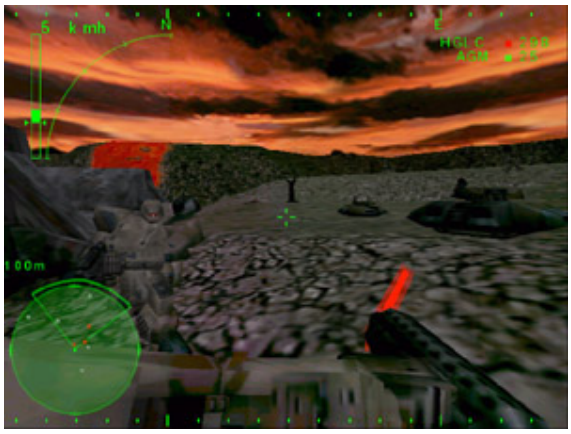
Our multiplayer system was crafted using a proven proprietary networking SDK developed at Activision. This reliance on preexisting technology allowed us to get multiplayer functionality in the engine and working very early on in the development cycle. Designing and integrating multiplayer functionality is often left until the end of a project, and that can create all manner of problems. Getting this level of complexity into our development schedule early probably saved the *Heavy Gear II* team an additional six months of work.

What Went Right

1. Effective prototypes

When I began working with Activision's *Heavy Gear II* production team, it was composed of a lead programmer, a top-of-the-line 3D graphics programmer, a director, a producer, a lead designer, and a lead artist. At that point, the team had just been given marching orders to produce a second prototype of the game for approval by the corporate brass. I was hired because this prototype required functional AI to give a feel of the intended game play. At this early stage, the game already looked super and a user-friendly 3D layout tool for the level designers was up and running. I couldn't believe the amount of work that these people put into producing the first prototype. Even more amazing was the fact that most, if not all, of this work was of the "keeper" variety - nearly all the code used in the green-light prototypes exists in the Darkside engine today. Our design tools were augmented in functionality, but the basic underlying technology driving these applications remained virtually unchanged through the entire development cycle.

2. The Darkside engine

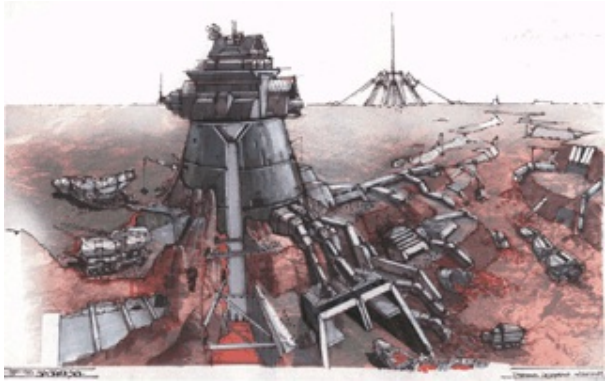


The Darkside engine's 3D math library was optimized for Intel's Katmai instruction set.

The Darkside engine created specifically for *Heavy Gear II* was an engineering gem. Although game players get a solid dose of its extraordinary graphics and animation capabilities, at its core it is nothing but a simple memory manager and leak tracker. This property of the engine allowed the programmers to track down and remedy most of the nastier bugs in the game long before it hit the QA floor. Beyond its concrete foundation, its modular design and expert use of C++ made it a snap to add or delete the different systems we wished to experiment with. Most of the foundation code and graphics subsystem was shared with our 3D layout tool, saving us a ton of extra work. This extensibility and proper use of the C++ language was invaluable as we faced a deluge of changing design requirements and additional game features. Our 3D math library was easily modified for Intel's Katmai Instruction Set, and this enabled our OEM group to strike up some valuable partnerships with external vendors. Although some *Heavy Gear II*-specific code may still lurk in the recesses of the Darkside, the engine's debug ability and modularity make it a novel choice for any 3D simulator Activision may wish to develop in the future.

3. They let us do our jobs

Those of you who have had the experience of a tough supervisor or management group breathing down your neck as you tiptoed down your critical path will appreciate this: our front-line governing body of director and producer did a superb job of trusting the professionals who worked under them. They made our team aware of upcoming milestones and the expectations of Activision's corporate group without succumbing to the temptations of micro-management and general megalomania. This held true even when the team hit roadblocks or missed milestones. They maintained their trust in us. This contributed to a fertile environment for new ideas and creative solutions from the team.



Mining installation on the wastes of Caprice

Activision's upper-level management also contributed to the game's success by letting the team decide when the game was finished. Management could have released the game early and announced a patch shortly thereafter, a practice that is becoming common with many

publishers nowadays. Instead, Activision gave game players a break and released a quality, bug-scarce title with a lot of replayability and immersion. This is an admirable goal in this era of market-driven development and patch-laden gaming web sites.

4. Great QA work

With Activision's business plan, its development teams enjoy the luxury of a highly organized QA department. On the front lines we have a smaller group known as "production testers." These folks deal directly with the development team on a daily basis and intercept a high percentage of the bugs before they ever make it out to external test groups and Activision's QA team proper. Production testers become very intimate with the inner workings of the game and are actually solicited for new ideas to improve the design. When the development team and production test team feel that the title has reached beta, Activision's main QA test group takes over for a formalized certification process. Concise reports are painstakingly generated to ensure that only genuine, replicable bugs ever make it back to the development teams. It is this group that is ultimately responsible for the release of the title, so they take their work very seriously.

Beyond Activision's in-house QA department, we were aided by an eager external group called "Visioneers" who avidly played any build we furnished for them. They reported all manner of problems, from hardware compatibility issues to boring game play. The size and diversity of this group made them an ideal QA avenue, as they represented a more accurate cross-section of the gaming community. Activision also invited people from different age groups to come into our office and play Heavy Gear II in an observational setting. Testers' responses were noted and given to the development team via written surveys. All these channels of software evaluation greatly streamlined the development process and contributed heavily to the quality of the finished product.

5. Good game pacing

If you sit down and play *Heavy Gear II*, you will notice a completely different feel from other games in the genre. The pace is faster and the action is much more compelling. This was a desired feature documented in the original specification of the game, but writing down that requirement in a document doesn't necessarily mean you will pull it off. At about the same time that *Heavy Gear II* was kicking off its production cycle, a series of first-person action shooters hit the streets. Most if not all of the team members were avid deathmatch players, and fierce competitions were held to decide which of us was the top dog. Subconsciously, much of the game-play feel we experienced while partaking of these frag-fests found its way into our title. A fun enemy AI unit was one that somehow evoked the same emotional response as the poor slob I had just fragged at lunchtime.

Early concept for the destruction of "Peace River"

This held true for the *Heavy Gear II* multiplayer experience as well. We tried to translate what we thought was fun and exciting in our lunchtime deathmatches into experience of fighting against hulking Gears. I'm not saying that if you want to create a great title with lots of excitement that you should play first-person shooters, but I do think that it is important for all game developers to be avid players as well. It is the game player that has the upper hand at identifying the abstract notion of "fun" and, isn't it the job of the developer to create that?

What Went Wrong

1. To demo or not to demo.

I love to play the demo version of any game before I buy it. In fact, as a gamer I purchase games based on how good the demo is. As a developer, however, a demo version of a game is a tricky thing to pull off. In our case, most of the game's critical systems were not functioning on all cylinders and many game assets didn't exist yet. We had to dedicate most of our time and resources to this Cimmerian task, although all our schedule called for was a small tiger-team consisting of a designer, programmer, and artist who worked on it a small percentage of their day for about two weeks.

When the smoke cleared and the demo was finally posted, the team gave a sigh of relief and basked in the warmth of a job well done. Unfortunately, this feeling quickly evaporated when we realized that we had taken almost a three-month departure from our original development schedule and totally exhausted ourselves in the process. I still believe a demo is important to the success of a game, but such a task should be closely correlated to the production of the main SKU. Attempting to factor the demo development time and resources into a schedule significantly different from the game itself can profoundly affect both the product's quality and timeliness.

2. Undocumented turnover.

This is a very common yet unpredictable aspect of game development. New and better jobs or personal issues always seem to snatch away even the most loyal and dedicated teammates. As a result, someone inherits the workload of the fallen comrade, putting the team and the schedule in a precarious situation.

When turnover occurred on the *Heavy Gear II* team, we and our schedule encountered a nasty surprise. Many of the systems we inherited were only partially implemented and virtually undocumented. To make things worse, much of the code was written to implement advanced animation and mathematical methods used all over the game engine, and we didn't have a technical design document that we could refer to in order to determine the intended solution for these systems. Working around the bugs and limitations in these systems cost the team even more time and generated bushels of frustration. This aspect of the turnover phenomenon is the least respected by developers and has the most profound and variable effect on scheduling. A periodic and thorough code review process is an effective way to defeat this problem.

3. Schedule too aggressive.

Heavy Gear II was built completely from the battleground up, so every aspect of the game required significant development time. Unfortunately, the development schedule was too optimistic about how long it would take to create the title.

Here's an idea of the scope of the game, as specified by our design document. *Heavy Gear II* required a brand new game engine and an accompanying suite of design, development, and debugging tools.

The game had more than 40 single-player missions, as well as numerous historical and instant-action missions. Additionally, multiplayer functionality (including cooperative play with multiplayer AI) was required for deathmatch, king of the hill, steal the beacon, and historical settings. We were also required to construct three different modes for each and all of these missions: terrestrial, space, and "Gomorrah" (combat in an enclosed, multi-level, near-future megacity) gaming modes.

Conceptual art for gateship "Celestrus" orbiting barren Caprice

The schedule called for a polished demo version of the game to be posted on all of the top gaming web sites. I joined Activision on November 10, 1997, after *Heavy Gear II*'s first prototype, and the final product was initially slated for release for Christmas of the following year. This period included a protracted QA run-through, effectively yielding a nine-month development cycle. Unfortunately, it was far too aggressive. Although we implemented all of the required features as defined in the original design specification (and many not included), we missed our final ship date of October 1998 by nine months.

4. Eleventh-hour solutions.

During the production of *Heavy Gear II*, many of our game designers and external testers began to notice a distinct absence of thrilling game play, which were attributed to several factors: the AI was too vicious, the AI-controlled squadmates were disobedient and difficult to deal with, and there was no noticeable ramp-up in difficulty and emotion — some missions were frightfully easy and others absolutely impossible to complete. (The average life expectancy of the player in some of our combat scenarios was about four seconds.)

The AI that controlled squadmates was deemed "disobedient" by some of the testers, and was difficult for the programming team to address.

Our design team felt that they didn't have proper tools and enough control over the AI and the environment to adequately tune their missions and make the game fun. Most of this was

due to the fact that we arrogant programmers designed and implemented most of the game logic without enough consultation and input from our able game design staff. To address these issues, our team had to depart totally from the design document and do a whole bunch of wild and creative thinking. Only after some ad hoc brainstorming sessions and grueling mission-by-mission playability tests did the pieces come together. Much of the kudos we received from the gaming community is directly related to these solutions.

However, a situation like this could have had far worse results. The next time around we will put much more thought and detail into our game design documents and provide a much more efficient education for our game design staff concerning game play manipulation and control via scripts. We will also give them a more prominent role in the initial design of these systems to enhance their understanding of the underlying technology.

5. Marketing issues.

After experiencing the high quality of our second prototype, people at Activision began to get very excited about the future of *Heavy Gear II*. We were given a highly favorable reception at E3 in 1998, where we finally revealed the game. We subsequently released a playable demo that met with similar acceptance. Our marketing staff, which had been pushing this title from the start, finally had the leverage it needed to differentiate *Heavy Gear II* from its competitors. Magazine articles began to run, OEM deals were made, and shelf displays were created as everyone anticipated the forthcoming release.

Only the development team knew how alarmingly behind schedule the game actually was, however. This was due in part because of some unfortunate turnover in our staff, but the main factor was the amount of time and effort we had allocated to creating the demo. We had hit roadblocks in the past and always rebounded in sterling fashion, so nobody on our team allowed themselves to believe that the demo would cause us to miss our target ship date.

Then we slipped. Suddenly we were plunged into a nearly interminable crunch mode. Our slip meant that the game wouldn't ship in time for the Christmas season, so we decided to shoot for a more realistic March release. Our marketing group did their best to respond to the blow and attempted to keep interest in *Heavy Gear II* alive. March came and went, and the release date became a dancing phantom beyond our reach. We developers knew we were close to completing the game, but nobody could give our marketing team a definite date so that they could keep the buzz up. Marketing did what it could to keep whetting the public's appetite as the weeks rolled by.

Missing the targeted ship date is a serious risk to teams that rely on new and unproven technologies — and it's especially perilous for teams working under compressed development schedules. Even the most innocuous development tasks, if underestimated or mishandled, can send your schedule flitting away beyond your control.

We Almost Got It Right…

We received great reviews from top gaming publications and web sites. Our marketing group piqued the interest of the gaming community and our development team produced a superior title. And since our development team had the luxury of starting this project from scratch, we had the opportunity to learn a wide range of new methods and techniques that would otherwise have remained beyond our reach.

Alas, our aggressive schedule and risk taking proved to be our Achilles' heel. Shipping a quality title is important, but so is strict adherence to budget and schedule. The *Heavy Gear II* team learned a great deal from this experience; we'll keep that with us for a long time. At least we left a solid and reusable game engine in our wake.

Beyond the role of volcanic guitarist and avid jai alai analyst, Clancy Imislund enjoys all aspects of engine design including 3D graphics, AI, tools, and multiplayer integration. For *Heavy Gear II*, he was responsible for the AI, scripting system, and tools development. Contact him at cimislund@activision.com.

Heavy Gear 2

Activision
Santa Monica, Calif.
(310) 255-2000
http://www.activision.com

Release date: July 1999

Intended platform: Windows 95/98

Project length: 19 months

Team size: 20

Critical development hardware: 3D-accelerated 200MHz PC

Critical development software: Softimage 3.7, Photoshop, Visual C++ 5.0, Visual SourceSafe, and numerous in-house tools

Return to the full version of this article