GAMASUTRA
The Art & Business of Making Games

PRINT

Postmortem: Redstorm's *Rainbow Six*

By Brian Upton

Rainbow Six and Red Storm Entertainment both came into being during the same week. When the company was formed in the fall of 1996, the first thing that we did was to spend a weekend brainstorming game ideas. That initial design session generated over a hundred possibilities that we then winnowed down to a handful that we thought had star potential. The only one that we unanimously agreed we had to build was HRT — a game based on the FBI's Hostage Rescue Team.

The Concept

It was a long road from HRT to *Rainbow Six*, but along the way, the basic outline of the title changed very little. We knew from the start that we wanted to capture the excitement of movies such as *Mission: Impossible* and *The Dirty Dozen* — the thrill of watching a team of skilled specialists pull off an operation with clockwork precision. We also knew that we wanted it to be an action game with a strong strategic component — a realistic shooter that would be fun to play even without a *Quake* player's twitch reflexes.

From that starting point, the title seemed to design itself. By the time we'd finished the first treatment a few weeks later, all the central game-play features were in place. We expanded the scope of game (rechristened *Black Ops*) beyond hostage rescue to encompass a variety of covert missions. Play would revolve around a planning phase followed by an action phase, and players would have to pick their teams from a pool of operatives with different strengths and weaknesses. Combat would be quick and deadly, but realistic. One shot would kill, but the targeting model would favor cautious aiming over the running-and-gunning that was typical of first-person shooters.

Ironically, the only major element that we hadn't developed during those first few weeks was the tie-in to Tom Clancy's book. Clancy was part of the original brainstorming session and had responded as enthusiastically as the rest of us to the HRT concept, but he hadn't yet decided to make it the subject of his next novel. Because we had moved away from doing a strict hostage rescue game, we batted around a lot of different Black Ops back stories in our design meetings, ranging in time from the World War II era to the near future. For a while, we considered setting the game in the 1960s at the height of the Cold War, giving it a very *Austin Powers*/*Avengers* feel. We eventually converged on the *Rainbow Six* back story in early 1997, but we didn't find out that we would be paralleling Clancy's novel until almost April. Fortunately, we'd been sharing information back and forth the whole time, so bringing the game in line with the book didn't involve too much extra work. (If you compare the game to the novel, however, you'll notice that they have different endings. Due to scheduling constraints, we had to lock down the final missions several months before Clancy finished writing. One of the pitfalls of parallel development…)

The Production

Originally, the *Rainbow Six* team consisted of me and one other programmer. Red Storm started development on four titles straight out of the gate, and all the teams were woefully understaffed for the first few months. The first *Rainbow Six* artist didn't come on board until the spring of 1997, with a full-time producer following shortly after. With such a small group, progress was slow. During that first winter and spring, all that we had time to do was throw together a rough framework for what was to follow. This lack of resources up front would come back to haunt us later.

Because we were so understaffed, we tried to fill the gaps in our schedule by licensing several crucial pieces of technology. The first was the 3D renderer itself. Virtus Corp., our parent company, was working on a next-generation rendering library for use in its own line of 3D tools. We decided to save ourselves work by building on top of the Virtus renderer, rather than developing our own. At first, this seemed to be an ideal solution to our problem. Virtus had been doing 3D applications for years, and the renderer that its engineers were working on was a very general cross-platform solution that ran well with lots of different types of hardware acceleration.

We also went out of house for our networking technology. We had researched a variety of third-party solutions, including Microsoft's DirectPlay, but we weren't satisfied with any of them. Just as we were on the verge of deciding that we'd have to write our own library, a local development group within IBM contacted us. The group's engineers were interested in finding uses for their powerful new Java-based client/server technology. The technology, called Inverse, was designed to allow collaborative computing between large numbers of Java applets. The IBM engineers wanted to see how it would perform in a number of different application domains, including games. Inverse supported all of the features that we wanted in a networking solution, such as network time synchronization and reliable detection of disconnects, so after much deliberation we decided to use it for *Rainbow Six*. Eventually, we would come to regret both of these third-party technology decisions, but not until months later in the project.

Over the summer of 1997, we acquired most of the motion capture data that was used for

animating the characters in the game. One of the advantages of working with Tom Clancy was that he put us in touch with a wide variety of consultants very quickly. Among the many experts we spoke with to get background information on counter-terrorism were two close-quarters combat trainers who worked for the arms manufacturer Heckler and Koch. When it came time to do our motion capture, these trainers volunteered to be our actors. They spent a couple of days at the Biovision studios in California being videotaped running through every motion in the game. Using real combat trainers for our motion capture data represented one of our better decisions. While a professional actor might have been tempted to overdo the motions for effect, these guys played it absolutely straight — the results are impressive. The game's characters come across as serious and competent, and are twice as scary as a result.

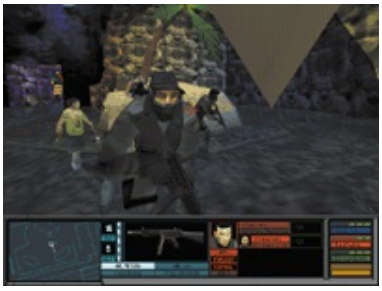
ion capture subjects made *Rainbow SIx's* character appear serious and competent.

Our crisis came in October of 1997. We'd been working hard all summer, but (although we refused to admit it) we were slipping further and further behind in our schedule. Partially, the delays were the result of my being completely overloaded. Partially, they were the result of the ambitious scale of the project: because the plot of Clancy's evolving novel was driving our level design, we'd committed ourselves to creating sixteen completely unique spaces — a huge art load. And partially, they were the result of the fact that the "time-saving" technology licenses that we'd set up were proving to be anything but.

Inverse was a great networking solution — for Java. Unfortunately, we wrote *Rainbow Six* in C++. Our initial research had suggested that mixing the two would be trivial. However, in practice the overhead involved in writing and debugging an application using two different languages at the same time was staggering. The interface code that tied the two parts together was larger than the entire networking library. It became clear that we'd have to scrap Inverse and write our own networking solution from scratch if we were ever going to get the product out the door.

(As a side note, we did continue to use Inverse for our Java-based products: last year's Politika and this year's Ruthless.Com. The problems we faced didn't arise from the code itself, but from mixing the two development environments.)

We also had problems with the Virtus rendering library. As we got deeper and deeper into *Rainbow Six*, we realized that if the game was going to run at an acceptable frame rate, we were going to have to implement a number of different renderer optimizations. Unfortunately, the Virtus renderer was a black box. It was designed to be a general-purpose solution for a wide variety of situations — a Swiss Army knife. With frame rates on high-end systems hovering in the single digits, we quickly realized that we would need a special-purpose solution instead.



The number of different renderer optimizations required for *Rainbow SIx* lead Redstorm to end it's reliance on third-party software.

In early November 1997, we put together a crisis plan. We pumped additional manpower into the team. We brought in Erik Erikson, our top graphics programmer, and Dave Weinstein, our top networking programmer, as troubleshooters. I stepped down as lead engineer and producer Carl Schnurr took over more of the game design responsibilities. The original schedule, which called for the product to ship in the spring, was pushed back four months. The artists went through several rounds of production pipeline streamlining until they could finally produce levels fast enough to meet the new ship date. Finally, we took immediate action to end our reliance on third-party software. We wrote an entire networking library from scratch and swapped it with the ailing Java code. Virtus graciously handed over the source code for the renderer and we totally overhauled it, pulling in code we'd been using on *Dominant Species*, the other 3D title that Red Storm had in progress at the time. All this took place over the holiday season. It was a very hectic two months.

From that point on, our development effort was a sprint to the finish line. The team was in crunch mode from February to July 1998. A variety of crises punctuated the final months of the project. In March, I came back on board as lead engineer when Peter McMurry, who'd been running development in my place since November 1997, had to step down for health reasons. As we added more and more code, builds grew longer and longer, finally reaching several hours in length, much to the frustration of the overworked engineers. The size of the executable started breaking all our tools, making profiling and bounds checking impossible. In order to make our ship date, we had to cut deeply into our testing time, raising the risk level even higher.

On the upside though, the closer we got to the end of the project, the more the excitement started to build. We showed a couple of cautious early demos to the press in March 1998 and were thrilled by the positive responses. (At this point, we were so deep into the product that we had no idea of what an outsider would think.) The real unveiling came at the 1998 E3 in Atlanta, Ga. Members of the development team ran the demos on the show floor — for most us, that was the longest stretch we'd had playing the game before it shipped. Almost all of the final game-play tweaks came out of what we learned over those three days.

What Went Right

1. A coherent vision

Throughout all of the ups and downs in the production process, *Rainbow Six*'s core game play never changed. We established early on a vision of what the final game would be and we maintained that vision right through to the end. I can't overstate the importance of this consistency. Simply sticking to the original concept saw the team through some really rough parts of the development cycle.

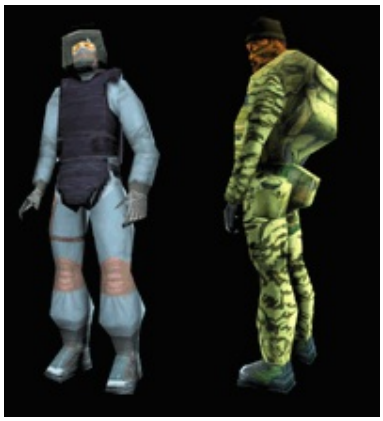
For one thing, this coherent vision meant that we were able to squeak by without adequate design documents. Many parts of the design were never written down, but because the team had a good idea of where we were headed, we were able to fill in many of the details on our own. Even when we had to perform massive engineering overhauls in the middle of the project, a lot of the existing art and code was salvageable.

Our vision also did a lot for morale. Many times we wondered if we'd ever finish the project, but we never doubted that the result would be great if we did. It's a lot easier to justify crunch hours when you believe in where the project is going.

2. An efficient art pipeline

The art team tried out four or five different production pipelines before they finally found one that would produce the levels that we wanted in the time that we had available. The problem was that we wanted to have sixteen unique spaces in the game — there would be almost no texture or geometry sharing from mission to mission. Furthermore, instead of creating our own level-building tool, we built everything using 3D Studio Max. Thus, artists had more freedom in the types of spaces that they could create, but they didn't have shortcuts to stamp out generic parts such as corridors or stairwells — everything had to be modeled by hand.

Eventually, the art team settled on a process designed to minimize the amount of wasted effort. Before anyone did any modeling, an artist would sketch out the entire level on paper and submit it for approval by both the producer and art lead. Then the modelers would build and play test just the level's geometry before it was textured. Each artist had a second computer on his desk running a lightweight version of the game engine so he could easily experiment with how the level would run in the game.



The wide variety of *Rainbow Six* characters were animated with motion capture data

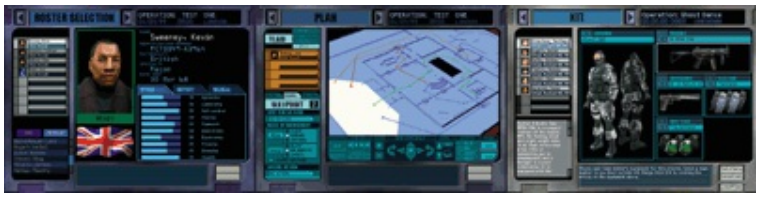
3. Tom Clancy's visibility

A good license won't help a bad game, but it can give a good game the visibility it needs to be a breakout title. When we first approached members of the gaming press with demos of *Rainbow Six* in the spring of 1998, they had no reason to take us seriously — we had no track record, no star developers, and no hype (O.K., not much hype…). We were showing a quirky title with a less-than-state-of-the-art rendering engine in a very competitive genre. With much-anticipated heavyweights such as *Sin*, *Half-Life* and *Daikatana* on the way, having Clancy's name on the box was crucial to getting people to take a first look at the title. Fortunately, the game play was compelling enough to turn those first looks into a groundswell of good press that carried us through to the launch.

4. Reworking the physics engine

In February 1998, we completely overhauled the *Rainbow Six* physics engine, which turned out to be a win on a variety of fronts. We'd retooled the renderer during the previous month, but our frame rate was still dragging. After running the code through a profiler, we figured that most of our time was going to collision checks — checks for characters colliding with the world and line-of-sight checks for the AI's visibility routines.

The problem was that every time the physics engine was asked to check for a collision, it calculated a very general 3D solution. Except in the cases of grenade bounces and bullet tracks, a 3D collision check was complete overkill. Over the next month, we reworked the engine to do most of its collision detection in 2D using a floor plan of the level. These collision floor plans would be generated algorithmically from the 3D level models.



A 2-D floor plan, created for the purpose of collision detection, also helped players in both the planning and action interfaces

The technique worked. In addition to getting the frame rate back up to a playable level, it also made collision detection more reliable. The game engine also used the floor plans to drive the pathfinding routines for the AI team members. Players would view these same floor plans as level maps in both the planning and action interfaces. By figuring out how to fix our low frame rates, we wound up with solutions to three or four other major outstanding engineering issues. Sometimes, the right thing to do is just throw part of the code out and start over.

5. Team cohesion

Red Storm employs no rock stars and no slackers. Everyone on the *Rainbow Six* team worked incredibly long hours under a tremendous

amount of pressure, but managed (mostly) to keep their tempers and their professional focus.

What Went Wrong

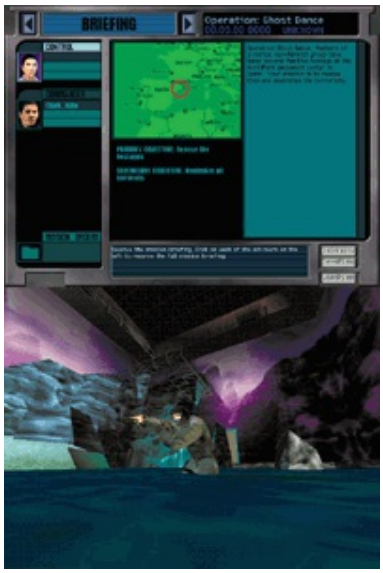
1. Lack of up-front design

We never had a proper design document, which meant that we generated a lot of code and art that we later had to scrap. What's worse, because we didn't have a detailed outline of what we were trying to build, we had no way to measure our progress (or lack thereof) accurately. We only realized that we were in trouble when it became glaringly obvious. If we'd been about the design rigorous up front, we would have known that we were slipping much sooner.

2. Understaffing at the start

This point is closely related to the previous point. Because we didn't have a firm design, it was impossible to do accurate time estimates. Red Storm was starved for manpower across the board, and because we didn't have a proper schedule, it was hard to come to grips with just how deep a hole we were digging for ourselves. There were always plenty of other things to do in getting a new company off the ground besides recruiting, and we were trying to run as lean as possible to make the most of our limited start-up capital. Given the circumstances, it was easy to rationalize understaffing the project and delaying new hires.

Additionally, I badly overestimated my own abilities. For Red Storm's first year, I was working four jobs: VP of engineering, lead engineer on *Rainbow Six*, designer on *Rainbow Six*, and programmer. Any one of these could have been a full-time position. In trying to cover all four, I spent all my time racing from one crisis to the next instead of actually getting real work done. And because I was acting as my own manager, there was no one to audit my performance. If one of the other leads was shirking his scheduling duties or blowing his milestones, I'd call him on it. But on my own project, I could always explain away what should have been clear warning signs of trouble.



***Rainbow Six's* game play involves a planning phase followed by an action phase.**

3. Reliance on unproven technology

Our external solutions for rendering and networking both fell through and had to be replaced with internally developed code late in the development cycle. In both cases, we were relying on software that was still under development. The core technology was sound, but we were plagued with inadequate documentation, changing programming interfaces, misunderstood performance requirements, and heavy integration costs. Because both packages were in flux, we failed to do a thorough evaluation of their limitations and capabilities. By the time it became obvious that neither was completely suited to our needs, it was too late to push for changes. In retrospect, we would have saved money and had a much smoother development process if we'd bitten the bullet early on and committed ourselves to building our own technology base.

The various mission levels called for the creation of sixteen completely unique spaces

4. Loss of key personnel

Losing even a junior member of a development team close to gold master can be devastating. When our lead engineer took ill in February 1998, we were faced with a serious crisis. For a few frantic weeks, we tried to recruit a lead from outside the company, but eventually it became obvious that there was no way we could bring someone in and get them up to speed in time for us to make our ship date in July 1998. Promoting from inside the team wasn't a possibility either — everyone's schedule was so tightly packed that they were already pulling overtime just to get their coding tasks done; no one had the bandwidth to handle lead responsibilities too.

Ultimately, I wound up stepping back in as lead. This time, however, we knew that for this arrangement to work I'd have to let my VP duties slide. The rest of management and the other senior engineers took up a lot of the slack, and Peter had set a strong direction for the project, so the transition went very smoothly. (After his health improved Peter returned to work at the end of the project, putting in reduced hours to finish off the *Rainbow Six* sound code.)

5. Insufficient testing time

We got lucky. As a result of our early missteps, the only way we could get the game done on time was to cut deeply into our testing schedule. We were still finding new crash bugs a week before gold master; if any of these had required major reengineering to fix, we would have been in deep trouble. That the game shipped as clean as it did is a testament to the incredible effort put in at the end by the engineering team. As it was, we still had to release several patches to clean up stuff that slipped through the cracks.

In the End…

Rainbow Six's development cycle was a 21-month roller coaster ride. The project was too ambitious from the start, particularly with the undersized, inexperienced team with which we began. We survived major overhauls of the graphics, networking, and simulation software late in the development cycle, as well as two changes of engineering leads within six months. By all rights, the final product should have been a buggy, unplayable mess. The reason it's not is that lots of very talented people put in lots of hard work.

I'm not going to say that *Rainbow Six* is the perfect game, but it is almost exactly the game that we originally set out to make back in 1996, both in look and game play. And the lessons that we've learned from the *Rainbow Six* production cycle have already been rolled into the next round of Red Storm products. Our current focus is on getting solid designs done up front and solid testing done on the back end — and on making great games, of course.

The *Rainbow Six* development team.

Rainbow Six

Red Storm Entertainment
Morrisville, N.C.
(919) 460-1776

http://www.redstorm.com

Release Date: August 1998

Intended Platform: Windows 95/98

Team Size: Sixteen full-time and six part time developers.

Critical Development Hardware: 400 MHz Pentium II w/64 MB RAM and a 3D accelerator.

Critical Development Software: Microsoft Visual C++, Sourcesafe, Hiprof, Boundschecker, and 3D Sutdio Max.

Brian Upton is the director of production design at Red Storm Entertainment. He as a Masters degree in computer science from the University of North Carolina at Chapel Hill and spent ten years as a graphics programmer before making the jump to full- time game

designer. He can be reached at brian.upton@redstorm.com.