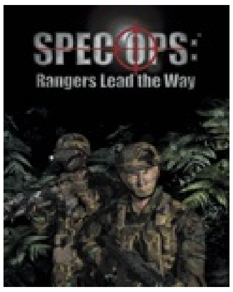
# Postmortem: Zombie's *SpecOps: Rangers Lead the Way*

By Wyeth Ridgway



Someone once said, "Experts are just people who have already made all the mistakes in their field." If this old saying is remotely true, I must be well on my way. After two years battling on the front lines at Zombie, I look back on the creation of *SpecOps* as a crusader looks back on victory in the Holy Land. Through the haze of past battles won and lost, I will now try to remember what it was that we did right and wrong, and how this product finally hit the shelves.

To begin, let's define the responsibilities of our game engine. Our game engine manages many aspects of the game: sound, graphics, physics, game core, and others. Taken alone, these components don't do anything — they're just tools. The game-specific AI uses these tools to perform logical game actions. In this manner, a complete game is composed of the game engine plus the game-specific AI.

As such, the *SpecOps* game is composed of the Viper game engine and the *SpecOps* AI and resources. In this article, I will primarily focus on the design aspects of the Viper engine, which was the first and most important step in creating *SpecOps*. In Table 1, you'll find a description of the project and its development environment. With these design requirements in mind, let's take a look at how each of the major components of the game engine was implemented.

**Game Core**

The game core provides an interface for all of the game engine components. Among other things, it defines and manages "objects" in the engine and allows them to be passed between components. For example, the game core might first send an object to the physics engine to be moved and then to the graphics engine to be drawn.

In the Viper engine, we chose to implement the objects hierarchically. Figure 1 shows a small portion of this hierarchy. The hierarchy allows memory optimizations because objects only require allocation of the structures that they use. It also provides some object-oriented structure to the program. In C, there was one .C file for each object type, with the same .H relationship as in Figure 1.

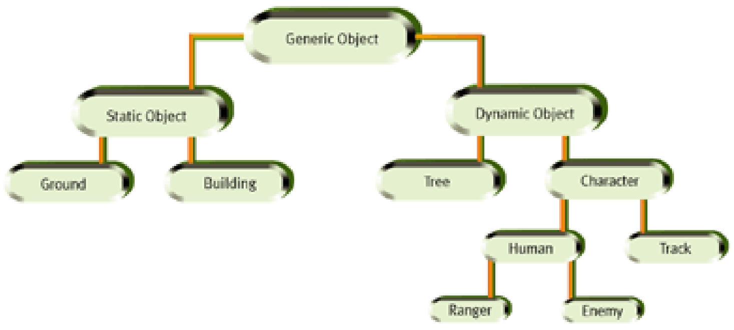


**Figure 1. Hierarchical object relationship from *SpecOps*.**

The only objects that the game core referenced and modified were those at the top of the hierarchy: Generic Object, Static Object, and Dynamic Object. The rest of the hierarchy is composed of game-specific data structures (such as the Character class). The support modules for these were separated from the rest of the code, so that it would be easy to remove them from the game engine code. This helped ensure the separation of game-specific logic from the game engine.

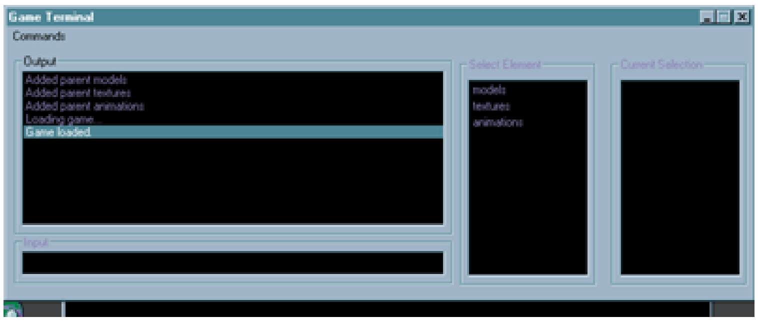
This design was expandable and easy to work with. While building the engine, we could separate out the *SpecOps*-specific code to create demos and even other games. Doing so at several points during the development cycle forced us to continually clean up any breaches of the design and maintain the reusability of the engine.

**Game Editor**

0ur game editor let us introduce new resources into the game engine, modify game play, debug game logic, and print out diagnostics. An editor environment needs to be easy to learn and use, so that the game designers and artists can make game modifications without a programmers being involved. It also needs to be continually maintained, providing access to new features of the engine as they are created and having bugs fixed or the interface modified to save time.

I wanted the Viper editor to be totally integrated into the game engine. This would help ensure that new game engine code didn't break the editor code, and would allow us to add new game engine features to the editor environment. I created an EDITOR compiler switch that created an encapsulating .DLL into the executable game file upon compilation. This .DLL ran the game as a child thread, allowing us to suspend the execution of the game and modify the contents of the game's memory. With this switch, we could activate functions in the game engine that we didn't want in the release version.

When active, the Viper editor displays a simple console . Command strings can be entered into this console to perform a variety of actions in the game engine. For instance, we would frequently use this functionality to load terrain geometry, create a binary space partition (BSP) tree of that geometry, and save the BSP tree to disk in the native file format.



```
Added parent models
Added parent textures
Added parent animations
Loading game...
Game loaded
```

```
models
textures
animations
```

**The Viper editor console.**

Overall, I was pretty happy with this editor. While it's more cryptic than the user interfaces of the *Quake* and *Unreal* editors, it has several benefits that those systems don't offer. First, all of our programmers were able to add functionality to it. Once our team learned how to use the editor's string commands (which took about five minutes), any programmer could add functions to support the code they wrote. Had I written a MFC-based GUI editor, I would have been supporting the editor on full-time basis because few team members were familiar with Windows code. Another point in favor of our editor was the fact that our geometry was being created separately by CAD tools, so we didn't need geometry creation facilities such as those found in the *Quake* and *Unreal* editors. Simply put, our needs were different from those games, and the command string interface fit our needs fairly well.

Yet, the editor did have drawbacks. Some tasks were difficult to complete using the command string interface. For example, to place dynamic objects (such as enemies and pickups) in the environment effectively, the game designers really needed a CAD-like, windowed interface that showed the entire level at various angles and allowed designers to place objects with mouse clicks. This could have been added fairly easily, but we lacked the time to implement it. As we advance the Viper engine, I'd like to see the editor seamlessly built into an existing CAD package, such as 3D Studio Max. That way, we could take advantage of an interface that the artists are already familiar with and draw upon pre-existing features.

---

**Resource Handling**

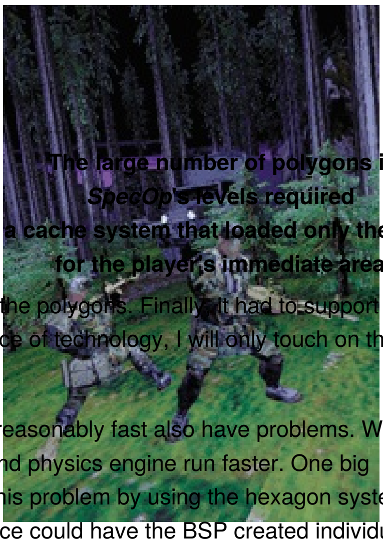
The Viper engine had several very difficult resource management issues to overcome. Each level in the game was composed of hundreds of thousands of polygons, which couldn't possibly be loaded at once; but we wanted the game's load time to be very brief. To accomplish this, we needed the game to page in and out of memory without affecting the frame rate, and we needed data loaded from disk to be ready to use with little or no additional processing.

To achieve these goals, we used the editor to load all the data into the game's native data structures and then wrote those structures directly to Viper's data file. These structures then loaded on demand at run time. Because we knew that this was a design requirement ahead of time, all the data structures in the game were designed to do this fairly easily. The only tricky part was when pointers were involved. We had to convert these pointers to offsets before they were saved to disk. At load time, Viper converted these offsets back to true pointers, and the data structures were ready to use.

Once we had a system that could efficiently load and use data, we had to design a cache system to load only those resources required for areas immediately around the player's position. We did this by dividing the world up into hexagonal pieces. At any time, three of these would be loaded in memory. The size of each hexagon was determined by how far the player could see in the environment (you don't want the player to be able to see the edge of the world). Each hexagon had to be further across than the viewing distance, yet small enough to efficiently load from the disk. The geometry for the hexagon was stored to disk, along with the resources (trees, pickups, enemy positions, and so on) associated with it. We completed the system by creating an asynchronous thread, which could load and unload the hexagons without halting the CPU. After a few optimizations (such as adjusting the hexagon size), we were able to load and unload these hexagons with only a one to two FPS impact.

Although the original design called for a similar system to handle the textures, we didn't have time to implement this feature. Instead, we simply loaded all the textures for a level at the start of the game, which worked out fairly well. With the additional RAM now available on 3D accelerators, it might even be a better solution anyway.

## Graphics and Animation


**The large number of polygons in *SpecOp*'s levels required a cache system that loaded only the data for the player's immediate area.**

The way in which Viper handles graphics and animation is one of our most beloved — and feared — components of the engine. More programmer time was spent on the graphics component than any other part of the engine. Viper had to deliver up to 10,000 polygons in a single frame (for reference, that's an entire *Quake* level), and still run at real-time frame rates. It also had to be able to run without a Z-buffer on the PlayStation, which required a sort routine for the polygons. Finally, it had to support software rendering as well as 3D hardware acceleration on the PC. Since this is such a huge piece of technology, I will only touch on the most prominent issues.

**BSPs.** By far the biggest bane of the 3D programmer is polygon sorting. Most methods that are reasonably fast also have problems. We chose to use 3D BSP trees because they're efficient to process, which helps both the graphics and physics engine run faster. One big problem with the BSPs, however, is that each one can take a long time to create. We mitigated this problem by using the hexagon system discussed previously. Since our world was already divided into non-overlapping pieces, each piece could have the BSP created individually. Since BSP creation is an exponentially complex problem, and our levels had hundreds of thousands of polygons, dividing the world into small pieces saved us from what would have been a feasibly insoluble problem. Furthermore, if and when we had to modify the world, we only had to recreate the BSP trees for the hexagons that had changed. Better yet, we could divide the labor of this task, so that we'd need each machine in the office to create just a couple of the BSP trees after hours. Thus, the artists could stay late and go through several iterations of BSPs in one night.

**Zombie used Z-buffering to handle moving objects for the software renderer.**

Another BSP-related problem was how to handle moving objects. Early on in the project, I was dismayed to hear that the guys at id Software had given up on solving this problem and had resorted to Z-buffering the dynamic objects in their *Quake* scenes. After several painful weeks of work, we finally had an acceptable solution, which exhibited sorting errors along the lines of Tomb Raider. Our solution was prohibitively slow, however, and when we reached our PC alpha stage and discovered that the PlayStation version was only running at five to ten FPS, we halted the development of that version. At that point, we switched to Z-buffering for the software renderer as well, since sorting errors are generally unacceptable on today's PC games.

A BSP solution is a mixed blessing. While they efficiently process algorithms, BSP trees don't handle dynamic objects well and they don't like to be modified at run time. Worse yet, the time required to create them can really slow up the game designers and artists. Because I won't be targeting a platform without Z-buffering again, I'm considering switching the Viper engine to a different data structure.

**Lighting.** Viper doesn't use light maps. When I debated the use of light maps versus an RGB vertex lighting scheme, the vertex approach seemed to be better supported by the hardware accelerators. I didn't like the time penalty of creating the light map surfaces, or the fact that the bus would be flooded with texture data. Viper's RGB lighting scheme supports an infinite number of colored light sources, combined at the vertex level and cleverly optimized to take almost no time penalty. The downside to this cheap lighting scheme is that it doesn't always have smooth edges along polygon borders. I'll probably dump this method in favor of something better in the near future, since processor speed is becoming so impressive.

**3D hardware cards.** I took a gamble and based all of Viper's development on the original 3Dfx Voodoo chipset. Two years ago, this chip had no marketshare, and cards based on it were more expensive than their competitors. However, 3Dfx had an excellent developer support group, and its board was fast and easy to use. Most importantly, it supported the basic polygon type that *SpecOps* would be based upon: Z-buffered, RGB-lit, textured, perspective-correct triangles. When I got the API and saw that you could start working with the board without writing any Windows code (through Glide), I knew I'd made the right choice. I wrote to Glide directly because it's easy and it's considerably faster than OpenGL or Direct3D. By the time you read this, Viper will be supporting other boards through a minimal subset of OpenGL.

**Importing geometry.** Viper can import geometry from Alias, Softimage, Lightscape, and 3D Studio MAX. Instead of supporting one package really well, we only had time to support all of them minimally. Still, there is something to be said for letting the artists work in the programs with which they're most comfortable. Also, at the time we were designing the engine, MAX and Softimage didn't support color vertex data, leaving us with few options. I think that we might soon work Viper into a single CAD package and rely on file format converters to move data around.

---

## Physics

The physics engine is the second most complex component of the game engine. It has to resolve all collisions, every frame, with minimal time overhead. Because the objects were all dealt with in a BSP tree, the time overhead was minimized, and collisions with the BSP are polygon accurate. Most of the Newtonian mechanics are true to life, although some things were simplified to save time.

Characters are implemented as a hierarchical models. As such, they're very time consuming to work with. Characters are often simplified to deformed spheres for collision purposes. This simplification can cause some strange side effects and doesn't allow bullets to hit specific locations on the body (this feature should have been added but alas, we ran out of time). Again, as processor speed increases and more work is offloaded onto dedicated 3D hardware, better physics simulation will become feasible.

Viper has a very impressive servo-based system for simulating vehicles. They are able to drive and fly over almost any terrain. The truck in the first level of *SpecOps* navigates the terrain based only on a series of vertices, which tells the servo where the road is. The truck applies a velocity in the direction in which it wants to drive, and the motion forward causes the wheels to move a corresponding amount. The friction model will cause the wheels to slip on steep slopes. A similar system of servos allows the helicopter to navigate over the trees in the forest and land in the clearings. The best part about working with servos is that they can deal with all sorts of environments without any additional work. The worst part is that they can sometimes do unpredictable things that are hard to reproduce.

***SpecOPs* characters were often simplified to deformed spheres for collision purposes.**

**Sound**

We created the sound engine with Microsoft's DirectSound. The initial implementation was fairly easy, but we spent months twiddling with it to deal with a variety of problems that popped up. Furthermore, we were using DirectX 3 when we began the project, but we shipped with DirectX 5. While there were supposedly no changes to the DirectSound API between these releases, the sound panning stopped working when we updated to version 5. Furthermore, because of some sort of multithreading issue, the sound never quite played correctly on Windows NT.

**Audio designers actually sampled the sounds of weapons being fired.**

In response to these DirectSound problems, we tried briefly to switch to DiamondWare's tools, which perform much better under Windows NT and have an easier-to-use API. Overall, DiamondWare Sound Toolkit was a better solution, but it also had a threading problem: it was monopolizing the bus and causing the 3D accelerator to hiccup. The company's technical support people said that they were aware of the problem and had no solution. In the end, we went back to DirectSound and lived with its problems.

With the spreading popularity of 3D sound hardware (such as Aureal), hopefully much of the sound mixing and spatial placement will be offloaded from the CPU. We strongly considered adding support for Aureal's A3D, either in a patch or an expansion pack for *SpecOps*.

**AI Engine**

Viper's game-specific AI is written entirely in a scripting language. The language's syntax looks like a cross between Basic and Lisp. The scripts describe hierarchical finite state machines and are object-oriented. The object-oriented implementation of these AI objects mirror the C implementation of the data structures in Figure 1. The scripts are compiled into a byte-code binary file that is executed at run time by the game engine. This was time consuming to implement, but provided us with several advantages over a comparable C implementation.

Most importantly, it created a hard boundary between the game-specific AI and the game engine. If you combined the main executable with different compiled script files and a separate set of resources, you would have a totally different game. This forced modularity saved us time and made it possible to make significant game AI changes late in development. The game's AI team was also able to work fairly independently from the game engine programmers. This was successful to the point that we were able to create numerous demos and even start production on a totally different title while the engine was still being built. In fact, I arrived at work one day to find that one of the game designers and one of the artists had gotten together and created a 3D monster truck racing demo without involving a programmer at all.

**The AI script language allowed the AI team to work fairly independently from the game engine programmers.**

This points out another benefit of using a script language: people without programming skills can modify the AI. Realistically, a programmer has to write 80 to 90 percent of any given script, but at that point the game designers can sit down and twiddle with it until it's just right. That said, I was often impressed at how adept at working with the script language many of the people at the office became. Our resident sound guy added nearly all of the sounds to the game with next to no assistance from the programming staff. I would often play the game on a Monday morning and be stunned by how much had been added without programmers being involved. Implementing game logic in C doesn't offer this.

Another benefit of using script-based AI was that because we had one compiled AI file per level, we only had the AI for a single level loaded at any given time. Not only did this separation make the division of labor easier, but it also saved a fair amount of memory.

All was not rosy, of course. There was the obvious time overhead for the extra level of indirection in the game AI. For the most part, the AI was so high-level that this overhead had no real impact on the game's performance. Still, more than once we had to implement a routine in C or optimize the run-time AI interpreter. Also, since this was the initial implementation of a complex system, we made a few design errors that had to be worked around. For the next title, we'll go back and address these changes.

The biggest problem with using a system like this was that the AI scripts were difficult to debug. Because they were largely just simple logic wrapped around calls to C functions, this problem was tolerable. Still, more than once we wished the scripts would just generate C code so that we could use Visual C++ to debug them. This might be a superior implementation, and something we will consider in the future.

---

**What Went Right**

Now that you have a good idea of how the Viper engine was implemented, I'll summarize the most prominent things that we did right and wrong. As is always the case at the end of a project, battles lost are always more prominent than battles won. If it were not for some key things that we did right, we might very well have failed.

**1. Hardware acceleration support**

Two years ago, publishers looked at 3D hardware accelerators with skepticism. Zombie made two key decisions at this point. First, we decided to fight tooth and nail to target a set of art specifically for hardware accelerators. Second, we made this art push the limits of the best accelerator on the market (at the time, the 3Dfx Voodoo 1). The result was that by the time the game shipped, there were many cards that were capable of running the game. In fact, even overshooting the graphics complexity by as much as we did, we weren't pushing the second generation of accelerators at all.

**2. Using a third-person perspective**

*SpecOps* was designed and under development for almost a year as a first-person game. When we set up a camera over the shoulder of the AI characters as a debugging tool, it was immediately clear that the game was meant to be third person. The sense of being a Ranger, the most important element of the game, was conveyed perfectly by seeing the character move through the environment as a Ranger. I thank Tomb Raider and Resident Evil for establishing this as a valid game interface. If we hadn't been exposed to these titles, I know we would have thought it was too risky to change perspective that late in the project.

**The third-person perspective effectively conveyed the sense of being a Ranger.**

**3. Outdoor environments**

We wanted this title to kill the flood of dungeon crawlers that have monopolized the market for years. We also wanted to present beautiful, realistic environments that demonstrated the computer's capabilities. Our levels were built to allow (and even encourage) the players to explore the world in which we placed them. Our first levels were more than two miles across, had nearly 100,000 trees, and required hours to traverse. We ended up scaling the levels back for playability purposes, but still managed to retain this concept. We also chose to create five totally different environments (forest, snow, jungle, desert, and city). This decision had two purposes: to create different tactical combat situations and to keep players' visual interest as they progress through the game. Kudos to games such as *Terra Nova* for helping us break down the walls of the dungeons.

**4. Complex mission objectives**

Games are based on levers and keys because they're easier to program. Most gamers just find this insulting to their puzzle-solving skills. As the lead programmer, part of me likes switches and keys. But as a game player, another part of me (the part that, admittedly, makes projects late) wants to constantly confront players with new and different challenges. The latter required painstakingly-created custom logic for every mission, with little reuse between levels. It also made debugging levels more painful than it might have been. Still, the end result was a big win for players. You will never quite get familiar with the game, because it will always throw something different at you. I hope more developers will pick up on this and stop building tools that spit out myriad games that all look and feel the same.

**5. Realism**

***SpecOps* Producer Sandra Smith on maneuvers with the U.S. Army Rangers.**

Whenever you're making a product that targets simulation fans, realism is key. The executive producer (a former Army Ranger) told us from the start that the product "was more like a movie than a game." With this as a premise, every detail in the game was researched and reproduced as perfectly as possible. We had Rangers come in for the motion capture sessions and photo shoots. We sampled sounds from the actual weapons used in the game. SOCOM (Special Operations Command) officers came in and reviewed our missions, and the game designers spent the better part of a year researching everything from environment characteristics to standard equipment carried by troops. Any one of these details might not have made much of a difference, but as a whole they brought the game up several notches. This attention to detail also created a very positive atmosphere for the entire team.

---

**What Went Wrong**

As much as some things sound like great ideas, they often aren't. Here are the top contenders for our biggest mistakes.

**1. Too much, too soon**

The spec for this game was more than a little overboard. It originally targeted the PlayStation, Macintosh, and PC. It had networking, supported 3D hardware, used motion capture, and contained support for just about everything else you could want in a game. It was also on a 15-month development cycle. To top it off, Zombie was writing the entire engine from scratch. On the up side, there was a healthy budget.

We dropped the Macintosh version right away, and we terminated the PlayStation version at alpha (when it was still running at five to eight FPS). Networking support was postponed for an expansion pack a year into development. While setting our sights so high was clearly the reason we got so far, dropping this many versions and features along the way was worse than anyone expected. I blame publishers as much as developers in this kind of situation. Publishers go through more product cycles than developers do, and should have some past experience telling them what is realistic to achieve. We learned the hard way what's possible to accomplish in 15 months; as a result, we completed the game in 20.

**2. Getting the team**

The last thing that I expected was that it would be hard to find good programmers in Seattle. I was hired at the start of *SpecOps*' development and didn't manage to bring the entire programming staff on board for nearly seven months. Needless to say, this caused substantial delays. We had similar problems hiring the art team. During the first months, the employees we hired burned out because they were trying to accomplish the duties of several people. I'm not sure what can be learned from this, but planning around such problems in the future will save some headaches.

**3. Losing the art leads**

About a month before E3 '97 rolled around, our art lead and a senior artist decided to leave the company. I had designed most of Viper's capabilities with them, and losing them at that critical time really devastated the project. With a little luck and some amazing dedication, we found people to fill in for them. We showed up at E3 last year with some wonderful art. Losing key staff just happens sometimes, and there's little that you can do about it but pick your chin up and wait for someone else to come along.

**4. Losing the publisher**

Right around E3 '97, we heard that our publisher, BMG Interactive, was planning to go out of business. It was not clear if our game would make it to the shelves, although BMG tried to assure us that everything was fine. After many months of things being up in the air, Ripcord Games came in and bought the title. In the interim, however, we lost morale, and there was some misdirection in our work effort. Ripcord came onto the scene so late that it had to really rush to get a marketing campaign going. Nobody wants something like this to happen, but it comes with the territory.

**5. Networking**

As mentioned earlier, at one point we mitigated being over schedule by dropping networking capabilities from the release. I don't think anyone actually believed that this was a good idea, but it somehow it happened anyway. The fact is that as much as 3D acceleration is the future, so is networking. Luckily, the gaming community has been taking it soft on us, and we're working hard to get out an expansion pack that has a variety of network play options.

That pretty much concludes the walkthough of the major components of the Viper engine and the problems we had while creating the title. While I could only really touch on the major issues we faced and the solutions we devised, hopefully it's enough to make the efforts of other developers a little easier. If there's a component of the Viper engine that you would like to see explained in further detail, please let me know. I encourage other developers to describe their projects in similar detail, so that we can learn from each other's mistakes as well as successes.

**Zombie's *SpecOps* development team.**

***SpecOps: Rangers Lead the Way***

Zombie Virtual Studios
114 1/2 First Avenue South
Seattle, WA, 98104
(206) 623-9655

http://www.zombie.com

**Core programming team:** 5; total contributing programmers: 13.

**Time in development:** 15 month development cycle, extended to 20 during development.

**Intended platform:** Initially targeted for both the PlayStation and the PC, with support for 3D hardware acceleration. Playstation dropped at alpha.

**Critical hardware:** P166 64MB machines with 3Dfx cards and two monitors.

**Critical software:** Visual C++ 5.0 and Sony's development tools.

**Notable technology:** Viper engine, which was created for this game

Return to the full version of this article