

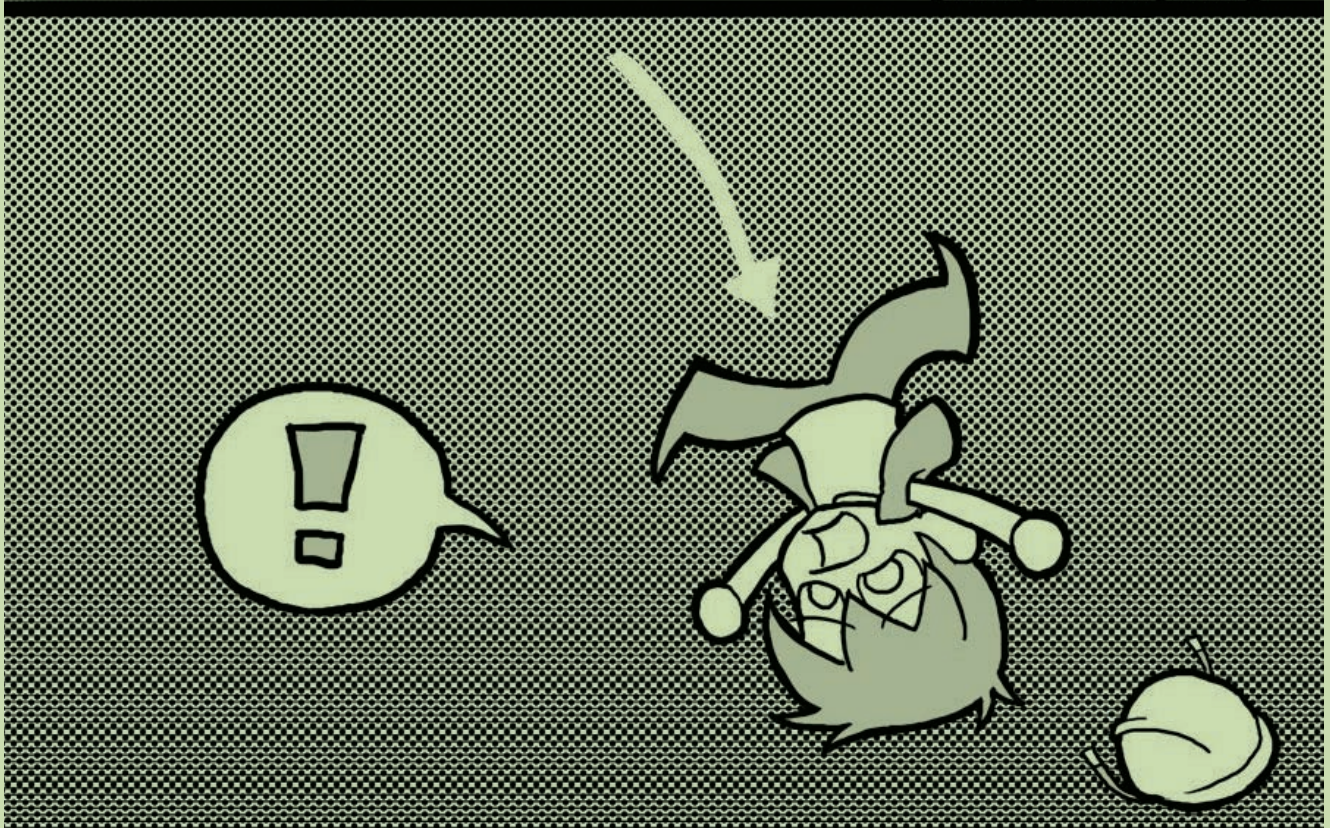
# DIRTY

## CODING TRICKS\*

////////// Programmers are often methodical and precise beasts who do their utmost to keep their code clean and pretty. But when the chips are down, the perfectly-planned schedule is shot, and the game needs to ship, “getting it done” can win out over elegance. In a case like this, a frazzled and overworked programmer is far more likely to ignore best practices, and hack in a less desirable solution to get the game out the door. We have here compiled nine testimonials from working developers, which chronicle times when they weren’t quite able to follow the script and had to pull some tricks to save a project. —Brandon Sheffield



CONTINUED ON PG 8



ILLUSTRATIONS BY JON "PERSONA" KIM

**\*THIS IS WHAT HAPPENS WHEN CODERS STOP BEING POLITE...AND START GETTING REAL**

CONTINUED FROM PG 7

## SHOOT ME FROM MY GOOD SIDE

**A**round four years ago I was working as a programmer on a multiplatform PlayStation 2, Xbox, and GameCube release. As development was coming to a close it should come as no surprise that some code hacks started creeping into the game to get the title shipped. The PS2 version in particular had a late, extremely hard to track down issue: A long soak test of the player character standing still in the first level of the game would cause it to periodically crash. Unfortunately, the crash was limited to disc-only retail builds that included no debugging information. With fears of a Sony technical requirement check (TRC) failure looming we worked hard to find a solution.

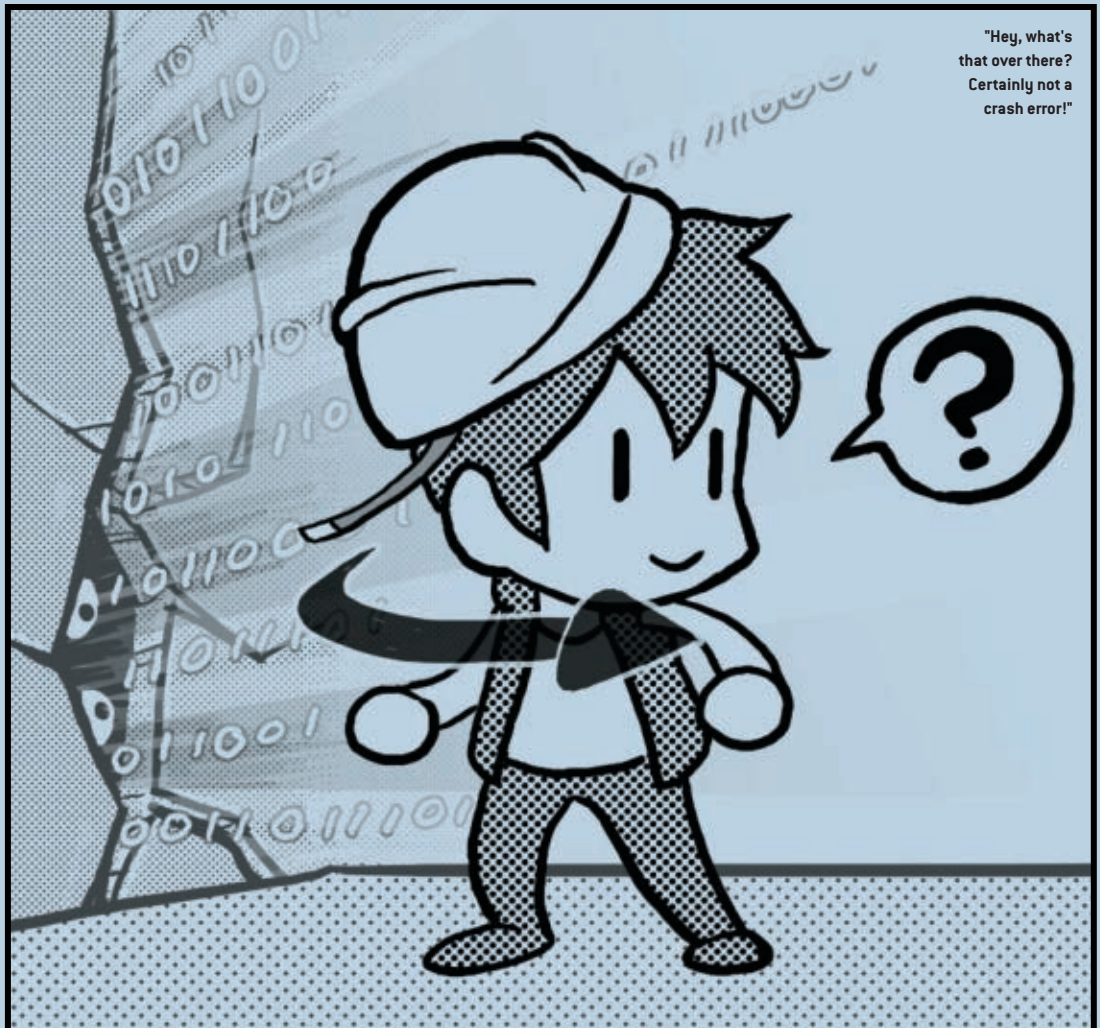
To narrow down the issue we continuously rendered a border around the edge of the screen with a different color for different sections of the code. For example, blue represented render setup, and green was the player control update. Because the crash was intermittent, the lead engineer on the title and I would manually burn a bunch of discs and start them up on an array of PS2s. (Keep in mind this was an independent developer, with

no IT team or fancy disc burning machines.) The test cycle—between making a code change, burning discs, deploying, and waiting to whittle down some approximation of where the crash was—took hours. It was the eleventh hour of development and there was a fixed number of these test cycles we could conduct.

Between stretches of passing the time over the weekend in the office with *WORLD OF WARCRAFT* while waiting for a crash, someone noticed that the game didn't crash if you rotated the camera 90 degrees to the right. Initially, the programming team correctly waved this off as some kind of fluke that was masking the underlying bug. This isn't the type of "fix" that would repair the root cause of a crash. That didn't stop us from shipping it though! As we ran out of time, we created the final PS2 disc with a rotated camera at level start—the other two platforms were already on track—and submitted everything. The game passed all the platform holders' TRC tests and was shipped to market on time.

I wouldn't call this a "coding trick," but it was definitely "dirty." The mysticism of the fix is disconcerting to this day, but thankfully I never heard of any reported user issues.

—Mark Cooke





## IDENTITY CRISIS

**T**his scene is familiar to all game developers: It's the day we're sending out the gold candidate for our Xbox 1 game. The whole team is playtesting the game all day long, making sure everything looks good. It's fun, it's solid, it's definitely a go in our minds.

In the afternoon, we make the last build with the last few game-balancing tweaks, and do one last playthrough session when disaster strikes: The game crashes hard! We all run to our workstations, fire up the debugger, and try to figure out what's going on. It's not something trivial, like an assert, or even something moderately hard to track down, like a divide by zero. It looks like memory is garbage in a few places, but the memory reporting comes out clean. What's going on?

One dinner and many hours later, our dreams of getting out on time shattered, we manage to track it down to one data file being loaded in with the wrong data. The wrong data? How's that possible? Our resource system boiled down every asset to a 64-bit identifier made out of the CRC32 of the full filename and the CRC32 of all the data contents. That was also our way of collapsing identical resource files into a single one in the game. With tens of thousands of files, and two years of development, we never had a conflict. Never.

Until now, that is.

It turns out that one of the innocent tweaks the designers had checked in that afternoon made it so a text file had the exact same filename and data CRC as another resource file, even though they were completely different!

Our hearts sank to our feet when we recognized the problem. There's no way we could change the resource indexing system in such a short period of time. Even if we pulled an all-nighter, there was no way to know for sure that everything would be stable in the morning.

Then, as quickly as despair swept over us, we realized how we could fix this on time for the gold candidate release. We opened up the text file responsible for the conflict, added a space at the end, and saved it. We looked at each other with huge grins on our faces and said:

"Ship it!"

—Noel Llopis

## DRIVING UNDER THE INFLUENCE

**I**'ve been tasked with taking care of a system dealing with vehicles, which is a fairly major part of our current game. Fortunately, we were able to grab the majority of the code from another studio. Unfortunately, the code isn't always perfect or well written, as is the perception with most code you don't write personally. I happened to find this nice bit of code that was simply trying to get a variable from the engine loop, but was going about it in the most backward way possible (See Listing 1).

The funniest part about this chunk of incredibly gross code is that there was a simple function on that object to get the frame count. Even if there weren't, the person who wrote this code could have easily added one himself! Needless to say, this code was not added to my game, nor to the game this code was taken from, as soon as I pointed it out. If this isn't an example of why code reviews are good, I don't know what is.

—Austin McGee

### LISTING 01

#### DRIVING UNDER THE INFLUENCE

```
//*****
//*****
// Function: AGameVehicle::Debug_GetFrameCount
//
//! A very hacky method to get the current frame
// count; the variable is protected.
//!
//! \return The current frame number.
//*****
//*****
UINTE AGameVehicle::Debug_GetFrameCount()
{
    BYTE* pEngineLoop = (BYTE*)&GEngineLoop;
    pEngineLoop += sizeof( Array<FLOAT> ) + sizeof(
    DOUBLE );
    INT iFrameCount = *((INT*)pEngineLoop);
    return iFrameCount;
}
```

## 10-TATIVE CODE

**B**ack at [company X], I think it was near the end of [the project], we had an object in one of the levels that needed to be hidden. We didn't want to re-export the level and we did not use checksum names. So right smack in the middle of the engine code we had something like the following. The game shipped with this in.

```
if( level == 10 && object == 56 )
{
    HideObject();
}
```

Maybe a year later, an artist using our engine came to us very frustrated about why an object in their level was not showing up after exporting to what resolved to level 10. I wonder why?

—Anonymous

## ALL SIGNS POINT TO "NO-NO"

**T**his issue came up during Raven Software's new WOLFENSTEIN, as I was setting up controller support for the 360. It turned out that for the Live integration, we had to know which controller was sending input events. The DOOM 3 input code we were using was derived almost straight from QUAKE 3, so it was a pretty simple system.

In the existing event system, each event was passed around with two integer arguments and a void pointer in case you needed extra parameters. So the goal was to associate a controller ID with each input event. No problem—just pack the controller ID into one of the event's integer arguments, right? Of course, they were both already in use.

Then came the next idea: Let's just use our fast no-fragmentation per-frame allocator to reserve some memory, plop the controller id in there, and then pass a pointer to it in the event's pointer slot.

It turns out that the event system would take it upon itself to `free()` the event's void pointer after processing the event. Of

course this was totally incompatible with the multi-heap approach that we were using. Moreover, there were a few legacy DOOM 3 chunks that actually relied on the event code calling `free()`, so we couldn't just remove the call without non-trivial changes across the codebase. What about adding a third integer parameter? Well, that would involve changing several hundred function calls.

The deadline is looming, I can't spend much more time on this. So, I did the unthinkable—I packed the controller id into the pointer parameter. I marked it as a horrible hack in a 4-line all-caps comment, and checked it in. This worked fine for a while, and lasted until the whole input system was replaced with something a little better down the line.

—David Dynerman

## VELCRO SNEAKS

Here's a tale from a project in the early PS2 days. We had a ton of collision/bounding issues that were largely solved at the last minute by a rewrite of character collision to use a "collider" model—a stack of spheres that were way better at resolving bumps vs. boxes than our hierarchical tree of oriented boxes ever could be (ah, the Land Before Havok). However, we had a rare bug for ages that was pretty maddening—we called it Velcro-ing—where every now and then the player character would be up against a wall and sliding along it, but then a sphere of the collider would suddenly decide it was on the wrong side of the wall and wouldn't let you get away from it (i.e. you'd be stuck to the wall's surface).

This was one of those god-awful "sounds easy" bugs—just figure out why the sphere thinks it's on the wrong side, and fix it. Problem is, you had to catch what happened in all the collision resolutions before it happened to figure this one out, and when you'd drop in things like handy conditional breakpoints to look for a weird response, or a push that made it get confused where it was on a skinny box or something similar, it'd bog the frame rate down and the issue simply wouldn't happen. (The real fix to this problem took blood, sweat, tears, and I think other fluids not usually required, but that is another story.)

Since we were trying to submit, and regularly would get this issue as a blocking bug, one "solution" was to buff out the bounding on whatever was causing the problem, which was guaranteed to make it not happen, but was clearly not the right fix. We'd forward the bug to our art team, they'd fix up that case, and in the ensuing turn-around time, we bought ourselves precious days to keep trying to track down the actual deep, subtle problem. Testers would typically send back a "player hit invisible wall, player must not hit invisible wall" bug of roughly equal severity (bounding the wall tightly again fixed that, as Velcro-ing was really rare and hard to reproduce anyway), but after a couple cycles of this as we cleaned up the rest of the game, the blood, sweat, and tears fix was eventually found and off we went to duplication and shelves.

This wasn't the proudest cycle of bug-turn-around-stalling, but it got the upper-ups off our backs, because we could say "oh, no, we released that to testing yesterday," and keep scrambling to figure out that damned flipping issue.

—Anonymous



## MEET MY DOG, PATCHES

There's an old joke that goes something like this:

Patient: "Doctor, it hurts when I do this."

Doctor: "Then stop doing it."

Funny, but are these also wise words when applied to the right situation? Consider the load of pain I found myself in when working on a conversion of a 3D third person shooter from the PC to the original PlayStation.

Now, the PS1 has no support for floating point numbers, so we were doing the conversion by basically recompiling the PC code and overloading all floats with fixed point. That actually worked fairly well, but where it fell apart was in collision detection. The level geometry that was supplied to us worked reasonably well in the PC version of the game, but when converted to fixed point, all kinds of seams, T-Junctions and other problems were nudged into existence by the microscopic differences in values between fixed and floats. This problem would manifest itself by the main character (called "Damp") simply falling through those tiny holes, into the abyss below the level.

We patched the holes we found, tweaking the geometry until Damp no longer fell through. But then the game went into test at the publisher, and suddenly a flood of "falling off the world" bugs were delivered to us. Every day a fresh batch of locations were found with these little holes that Damp could slip through. We would fix that bit of geometry, then the next day they would send us ten more. This went on for several days. The publisher's test department employed one guy whose only job was to jump around the world for ten hours a day, looking for places he could fall through.

The problem here was that the geometry was bad. It was not tight and seamless geometry. It worked on the PC, but not on the PS1, where the fixed point math vastly magnified the problems. The ideal solution was to fix the geometry to make it seamless.

The many faces of Nick Waanders.



However, this was a vast task, impossible to do in the time available with our limited resources, so we were relying on the test department to find the problem areas for us.

The problem with that approach was that they never stopped finding them. Every day bought more pain. Every day new variants of the same bug. It seemed like it would never end.

Eventually the penny dropped. The *real* problem was not that the geometry had small holes in it. The problem was that Damp fell through those holes. With that in mind, I was able to code a very quick and simple fix that looked something like:

```
IF (Damp will fall through a hole()) THEN
  Don't do it
```

#### LISTING 01

### MEET MY DOG, PATCHES

```
damp_old = damp_loc;
move_damp();
if (NoCollision())
{
  damp_loc = damp_old;
}
```

#### LISTING 03

### MEET MY DOG, PATCHES

```
if (damp_aliencoll != old_aliencoll &&
strcmpi("X4DOOR",damp_aliencoll->enemy->ename)==0 &&
StartArena == 6 && damp_loc.y<13370)
{
  damp_loc.y = damp_old.y; // don't let damp ever
touch the door.. (move away in the x and y)
  damp_loc.x = damp_old.x;
  damp_aliencoll = NULL; // and say thusly!!!
}
```

The actual code was not really much more complex than that (see Listing 2).

In one swoop a thousand bugs were fixed. Now instead of falling off the level, Damp would just shudder a bit when he walked over the holes. We found what was hurting us, and we stopped doing it. The publisher laid off their “jump around” tester, and the game shipped.

Well, it shipped eventually. Spurred on by the success of “if A=bad then NOT A”, I used this tool to patch several more bugs—which nearly all had to do with the collision code. Near the end of development the bugs became more and more specific, and the fixes became more and more “Don’t do thispreciseandexacting” (see Listing 3, actual shipped code).

What does that code do? Well, basically there was a problem with Damp touching a particular type of door in a particular level in a particular location, rather than fix the root cause of the problem, I simply made it so that if Damp ever touched the door, then I’d move him away, and pretend it never happened. Problem solved.

Looking back I find this code quite horrifying. It was patching bugs and not fixing them. Unfortunately the real fix would have been to go and rework the entire game’s geometry and collision system specifically with the PS1 fixed point limitations in mind. The schedule was initially aggressive, so we always seemed close to finishing, so the quick patch always won over the comprehensive, expensive fix. But it did not go well. Hundreds of patches were needed, and then the patches themselves started causing problems, so more patches were added to turn off the patches in hyper-specific circumstances. The bugs kept coming, and I kept beating them back with patches. Eventually I won, but at a cost of shipping several months behind schedule, and working 14 hour days for all of those several months.

That experience soured me against “the patch.” Now I always try to dig right down to the root cause of a bug, even if a simple, and seemingly safe, patch is available. I want my code to be healthy. If you go to the doctor and tell him “it hurts when I do this,” then you expect him to find out *why* it hurts, and to fix that. Your pain and your code’s bugs might be symptoms of something far more serious. The moral: Treat your code like you would want a doctor to treat you.

—Mick West



## YOU WOULDN'T LIKE ME WHEN I'M ANGRY

I once worked at THQ studio Relic Entertainment on *THE OUTFIT*, which some may remember as one of the earlier games for the Xbox 360. We started with a PC engine [single-threaded], and we had to convert it to a complete game on a next-gen multi-core console in about 18 months. About three months before shipping, we were still running at about 5 FPS on the 360. Obviously this game needed some severe optimization.

When I did some performance measurements, it became clear that as much as the code was slow and very “PC,” there were also lots of problems on the content side as well. Some models were too detailed, some shaders were too expensive, and some missions simply had too many guys running around. It’s hard to convince a team of 100 people that the programmers can’t simply “fix” the performance of the engine, and that some of the ways people had gotten used to working needed to be changed. People needed to understand that the performance of the game was everybody’s problem, and I figured the best way to do this is with a bit of humor that had a bit of hidden truth behind it.

The solution took maybe an hour. A fellow programmer took four pictures of my face—one really happy, one normal, one a bit angry, and one where I am pulling my hair out. I put this image in the corner of the screen, and it was linked to the frame rate. If the game ran at over 30fps, I was really happy, if it ran below 20, I was angry. After this change, the whole FPS issue transformed from, “Ah, the programmers will fix it.” to, “Hmm, if I put this model in, Nick is going to be angry! I’d better optimize this a little first.” People could instantly see if a change they made had an impact on the frame rate, and we ended up shipping the game at 30fps.

—Nick Waanders

## THE PROGRAMMING ANTIHERO

I was fresh out of college, still wet behind the ears, and about to enter the beta phase of my first professional game project—a late-90s PC title. It had been an exciting rollercoaster ride, as projects often are. All the content was in and the game was looking good. There was one problem

though: We were way over our memory budget.

Since most memory was taken up by models and textures, we worked with the artists to reduce the memory footprint of the game as much as possible. We scaled down images, decimated models, and compressed textures. Sometimes we did this with the support of the artists, and sometimes over their dead bodies. We cut megabyte after megabyte, and after a few days of frantic activity, we reached a point where we felt there was nothing else we could do. Unless we cut some major content, there was no way we could free up any more memory. Exhausted, we evaluated our current memory usage. We were still 1.5 MB over the memory limit!

At this point one of the most experienced programmers in the team, one who had survived many years of development in the “good old days,” decided to take matters into his own hands. He called me into his office, and we set out upon what I imagined would be another exhausting session of freeing up memory.

Instead, he brought up a source file and pointed to this line:

```
static char buffer[1024*1024*2];
```

“See this?” he said. And then deleted it with a single keystroke. Done!

He probably saw the horror in my eyes, so he explained to me that he had put aside those two megabytes of memory early in the development cycle. He knew from experience that it was always impossible to cut content down to memory budgets, and that many projects had come close to failing because of it. So now, as a regular practice, he always put aside a nice block of memory to free up when it’s really needed.

He walked out of the office and announced he had reduced the memory footprint to within budget constraints—he was toasted as the hero of the project.

As horrified as I was back then about such a “barbaric” practice, I have to admit that I’m warming up to it. I haven’t gotten into the frame of mind where I can put it to use yet, but I can see how sometimes, when you’re up against the wall, having a bit of memory tucked away for a rainy day can really make a difference. Funny how time and experience changes everything. ❖

—Noel Llopis