

STUDYING SPEEDRUNNERS

DEBUG DOOM BY WATCHING ITS TOP PLAYERS AT WORK

Whether you've just written your first "Hello, World!" or you've been programming since the punch-card days, you know that we all make mistakes as a necessary part of the learning process. Sometimes we catch those mistakes in time, and other times those mistakes end up lingering in the final shipped product, becoming part of the game whether we like it or not. In this article, we're going to dissect a few bugs found in DOOM, id Software's seminal first-person shooter—bugs that would be very easy to miss if it weren't for the devoted speedrunners who exploited them to move significantly faster than its developers intended.

SPEEDRUNNING DOOM For the uninitiated: Speedrunners are players who specialize in playing through games as quickly as possible. Naturally, they're concerned with looking for any strategy that allows you to move faster, whether they're engine-wide exploits or specific paths through certain levels. We're going to focus on a bug in DOOM that is a speedrunner's dream: a simple glitch in the movement code that allows us to move much faster than intended.

This has broad implications. Moving faster means jumping higher and farther (you can't jump in DOOM, but a faster speed still lets you cross wider gaps than normal—see **Figure 1**). Some levels intend you to go out of your way to find a switch to raise a bridge, for instance. With a trick that lets you move faster, you might be able to



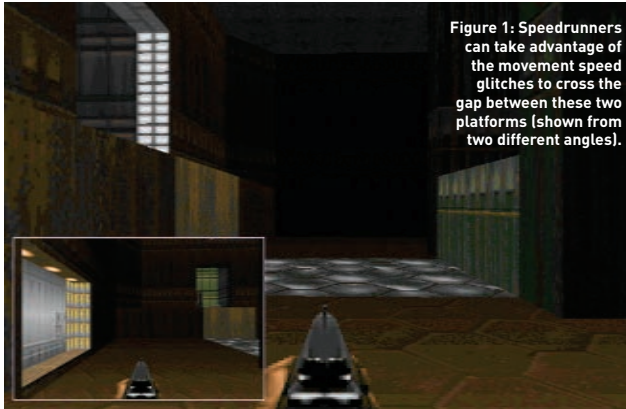


Figure 1: Speedrunners can take advantage of the movement speed glitches to cross the gap between these two platforms (shown from two different angles).

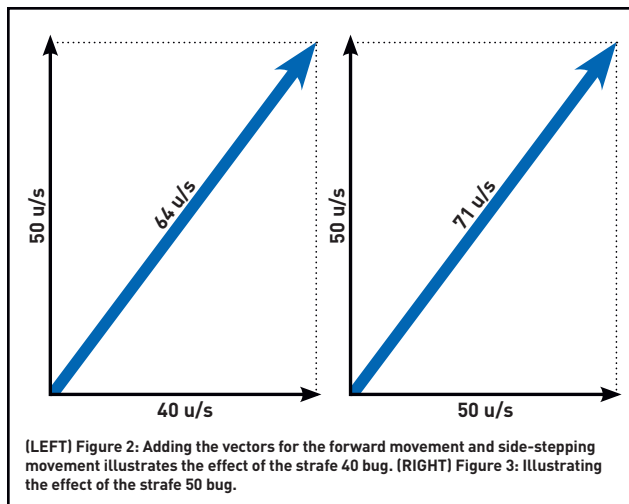
cross the gap without the platform, which means skipping much of the level.

We will look at three movement bugs in particular:

- Strafe 40: Moving forward while also strafing in either direction will allow you to move 28% faster.
- Strafe 50: Moving forward while also strafing in either direction and also toggling a feature that interprets turning as strafing allows you to move 41% faster.
- Wallrunning: Moving along a wall of a particular orientation will move you at almost twice the normal speed.

Go watch a few DOOM speedruns (you can find a few at the Speed Demos Archive: <http://speeddemosarchive.com/Doom.html>). Pay careful attention to the direction the player tends to face. Most of the time, the player is not running in the same direction they are facing, but at a slight diagonal. Essentially, they are moving both forward and sideways at the same time by pressing two movement keys at once—in a modern first-person shooter, this would be like pressing W and A simultaneously. In DOOM, when you use both movement keys while running, the game erroneously moves your character at 128% the normal speed. This is called the strafe 40 glitch.

KNEE-DEEP IN THE CODE Let's now investigate why these particular bugs happen. DOOM is written in C, which is a very low-level programming language, meaning it is designed for speed rather than programmer comfort. This was a very common language for games in the '90s, since the language was designed



to allow you to write very optimized code and push the limits of the technology of the time. With that in mind, I have pulled out relevant pieces of the actual source (<https://github.com/id-Software/Doom>) with very little changes to make the code more clear.

Turning our attention to the DOOM source: There are several functions that deal with moving the player. The gist of them is that for each frame, the game decides how to move the player before it draws the screen. The game looks at which keys are pressed, and if the "walk forward" key is down, it will update the player's position to move them forward. It may then look at handling collisions and projectiles, but we're going to focus on the movement part.

In order to trigger the strafe 40 bug, all you need to do is move forward and strafe at the same time. We can find out why this happens in **Listing 1**—refer to the `P_MovePlayer` function. The `cmd` element holds the distances to move per frame. `cmd>forwardmove` and `cmd>sidemove` contain the distances to travel, either ahead of the player or to the side of the player, respectively. It will set an on-ground value to "true" if the player's z position (their distance from the ground) matches that of the floor the player is currently over. Therefore, it only wants to move if the player is in contact with the ground.

Given that the player is on the ground, the code checks to see if the player is due to move forward (`cmd>forwardmove` will be non-zero) and then calls another piece of code that simply repositions the player to reflect that movement. It does the same thing for a strafe, except in a different direction.

From here, we can see the mistake. We can move forward or strafe independently, and it would work as expected. However, if we move forward and strafe, the player will thrust forward, and then afterward, thrust sideways. From the perspective of the game, these two movements are done at the same time, because both are performed before the screen is drawn and the enemies react. Therefore, the actual speed is given by the sum of the vectors; that is, the length of the hypotenuse as illustrated in **Figure 2**.

Of course, there are many ways to repair this bug and handle movement more correctly. One way would be to determine the angle of the movement and always use the same distance instead of positioning the player twice. Instead of moving in the player's direction and then moving again in another direction for the strafe, simply calculate the movement angle (around 50 degrees for walking and strafing), and `P_Thrust` only once in that direction.

Notice that the code does not account for which direction you are strafing. This is because of a naive optimization: The distance (in the code, this is the `cmd>forwardmove * 2048`) you give to `P_Thrust` can be negative to move in the opposite direction. For the fix, you will have to account for the direction in which you are strafing to get the correct angle, but now you always give a positive distance.

FIXING STRAFE 50 To understand how to exploit the strafe 50 bug, we have to look at how it decides `cmd>forwardmove` and `cmd>sidemove`. These values determine how many units the player will travel per frame in each of those two directions. The flaw is that you can artificially affect these values by having the game accidentally count two different keys as movement during a single frame.

Basically, you tell it to move you to the right... twice, and it diligently listens to you. For this, let's look at the input handling code and the function `G_BuildTiccmd` in **Listing 2**.

We can see the familiar `cmd>forwardmove` and `cmd>sidemove` at the bottom; this chunk is the code that determines those, and we're going to explore how it translates the player's keypresses into in-game values.

In DOOM, you can strafe one of two ways: Either you use a modifier key that causes your "turn left" and "turn right" keys to change function to "strafe left" and "strafe right" while it's depressed, or you assign dedicated strafe key for each direction, which works the same way as the A or D keys on most modern games.

With that in mind, look at the code. By the "**Section 1**" comment in **Listing 2**, we see that the game looks to see if that strafe toggle is held. Depending on whether the toggle is held or not, DOOM either stores the `cmd>angleturn`, which tells the

LISTING 1 Dissecting the strafe 40 bug in the player movement code

```

void P_MovePlayer (player_t* player) {
    ticcmd_t* cmd;
    cmd = &player->cmd;

    // Turn the player
    player->mo->angle += (cmd->angleturn<<16);

    // Do not let the player control movement
    // if not onground.
    onground = (player->mo->z <= player->mo->floorz);

    // Move the player forward, if allowed
    if (cmd->forwardmove && onground)
        P_Thrust (player, player->mo->angle, cmd->forwardmove*2048);

    // Move the player sideways, if allowed
    if (cmd->sidemove && onground)
        P_Thrust (player, player->mo->angle-ANG90, cmd->sidemove*2048);
}

```

LISTING 2 The Doom code responsible for the strafe 50 bug

```

void G_BuildTiccmd (ticcmd_t* cmd) {
    boolean strafe;
    int speed;
    int forward;
    int side;

    // We are strafing if a strafe key is pressed
    strafe = gamekeydown[key_strafe];

    // Is the run key pressed?
    speed = gamekeydown[key_speed];

    // The distances we are moving are initially zero
    forward = side = 0;

    // Determine distances to move
    if (strafe) {
        // (Section 1)
        // If the strafe toggle is on, interpret moving left and right
        // as strafing left and right.
        if (gamekeydown[key_right]) // Strafe right
            side += sidemove[ speed ];
        if (gamekeydown[key_left]) // Strafe left
            side -= sidemove[ speed ];
    }
    else {
        if (gamekeydown[key_right]) // Move right
            cmd->angleturn -= angleturn[ speed ];
        if (gamekeydown[key_left]) // Move left
            cmd->angleturn += angleturn[ speed ];
    }

    if (gamekeydown[key_up]) // Move forward
        forward += forwardmove[ speed ];
    if (gamekeydown[key_down]) // Move backward
        forward -= forwardmove[ speed ];

    // (Section 2) Strafe right
    if (gamekeydown[key_straferight])
        side += sidemove[ speed ];

    // Strafe left
    if (gamekeydown[key_strafeleft])
        side -= sidemove[ speed ];

    // (Section 3) Cap speed
    if (side > forwardmove[ speed ])
        side = forwardmove[ speed ];
    else if (side < -forwardmove[ speed ])
        side = -forwardmove[ speed ];

    cmd->forwardmove += forward;
    cmd->sidemove += side;
}

```

P_MovePlayer function above to turn the given degrees before drawing, or it completely ignores the turning and instead strafes by adding a distance to move (affected by whether or not run is enabled) to the variable `side`, which is initially zero.

So: We know that when we have the strafe toggle on and we press the right arrow key, it will handle that as a strafe to the right and add some distance to the variable `side`. Now take note of the “**Section 2**” comment in **Listing 2**. This code happens independently of the strafe toggle. If you also press the strafe right key, this code will add even more distance to the current value (Note: `side+=sidemove[speed]` is the same as writing `side=side+sidemove[speed]` in C). That means if we press the dedicated strafe right key and we also press right while the strafe toggle is on, then we will effectively strafe twice!

Interestingly, the programmer doesn’t seem very optimistic about the code; if you look at the “**Section 3**” comment in **Listing 2**, you’ll notice that the speed of side movement is capped to the maximum speed you can run forward. However, since strafing was intended to be slower than running forward at only 40 units per second, this code incorrectly caps the sideways strafe speed to 50 units per second! **Figure 3** shows how our movement is now calculated. Since this does not interfere with the strafe 40 bug we investigated earlier, we just found a way to make it more effective.

Even though this bug seems more severe and tricky, it’s far easier to solve than strafe 40—all you need to do is put the code in **Section 2** into the else block after **Section 1**, such that the normal strafe is only considered if strafe toggle is off. Alternately, if you want to allow for strafe to be pressed by the dedicated strafe key even if strafe toggle is on, just fix the code in **Section 3** by capping the player’s sideways movement speed to the proper maximum strafe movement speed (40 units per second). You will now not be able to do any better than the original strafe 40.

FASTER THAN ROCKETS:

WALLRUNNING If you thought the strafe 40 and strafe 50 bugs were handy for speedrunners, this next one is even bigger: When you encounter a wall of a particular orientation, running against this wall propels you to nearly twice the already-quick strafe 50 speed.

In the last two sections, we have looked at the code that handles input and determining the player position. There is nothing left to discover in this code that would yield this bug, so we’ll need to look elsewhere. Specifically, we must shift our focus away from how a player moves, but what stops them from moving.

In the real world, we know that an object in motion stays in motion until it

Scotland. Famous for golf and innovative games development.

All the best players
come here.



We've got quite a reputation for invention, innovation and discovery. And it stretches way beyond Highland Games, the bicycle and golf. We were the first to award a degree in Computer Games Technology and our pioneering games work ranges from the creation of Grand Theft Auto to Bloons and Quarrel. The fact is, Scotland is one of Europe's top games development locations. We have a growing hive of creative and talented games developers and our universities are

developing new and converging technologies across a range of platforms.

Above all, our people are dedicated, committed and passionate for success. And this passion, combined with our world-class academic institutions, outstanding research and superb facilities make Scotland financially irresistible. We can develop your products and help shape your business. And that's what makes Scotland such a popular place to live, work and play.

To see what we can do for your business, visit www.sdi.co.uk/games

SCOTLAND. SUCCESS LIKES IT HERE.



interacts with another body. In the virtual world, we cannot always exactly recreate this phenomenon, but we can simulate this idea by making use of very simple collision-detection algorithms.

We start with a moving object with some sort of trajectory. In the movement code above, we determined the new position of the player, so the trajectory is simply the line drawn from the old position to this new one. With this trajectory, all we must do is check if this line intersects an object or a wall, which just requires some basic algebraic geometry.

So what happens when the moving object hits a wall? It may seem reasonable to say that the wall impedes your movement, so you end at the intersection of your trajectory and the wall. You stop dead, so to speak. However, an action game developer wants the player to always feel like they are moving, and a dead stop wouldn't produce that effect, so the game needs to simulate some momentum. But how do we want to model this? What happens when you hit a wall at an angle?

In DOOM, if you run into a wall at an angle, you don't stop so much as slide along it. The wall cannot contain all of the energy, and so you keep a bit of your velocity and move along the wall only coming to a stop as the result of friction. As you may be able to gather, there are a lot of factors that determine how far you slide and in which direction, while avoiding the proper, yet computationally expensive physics calculation proves to be a tricky problem.

Let us now review the code in **Listing 3**. The reasoning behind this bug has been, within the speedrunning scene at least, a bit of a mystery, so let's figure out why this happens.

To put the prior code in perspective, DOOM goes through two stages before it renders. The first stage is the one we have reviewed for the strafing bugs. In this stage, DOOM determines the new positions of all objects, including the player. The next stage will then review all of these trajectories and determine final placements of objects due to collisions. This is the stage we are concerned with now.

Check out the **P_XYMovement** function in **Listing 3**. This piece of code handles the update of the player's x and y coordinates, which correspond to the player's position with respect to the floor. The remaining z coordinate, handled elsewhere, is the player's height. The function computes the momentum of the player and uses this to see if the player will collide with walls or objects using the **P_TryMove** function. As you can see, if the player cannot move to the given position, it will call **P_SlideMove**, which will move the player to a position along the wall given their current momentum.

Can you spot the mistake? Hint: It has to do with the purpose of the loop itself. It seems that the programmer was very pessimistic about the collision detection code, and decided that when the player is moving very quickly, the collision detection function **P_TryMove** should be called twice: Once at half the distance, and then again for the rest. The idea is that this improves correctness since you will be less likely to skip over smaller triggers. (Ironically, this attempt to avoid one bug caused another one.)

This is a problem because the "xmove" and "ymove" variables are divided in half to ensure the loop runs twice, however, the **P_SlideMove** function receives the untouched original momentum structure in "mo." Therefore, when running full speed (strafe 40)



Figure 4: Wallrunning lets you survive the run through this moat, which is much faster than beating the level in the intended manner.



LISTING 3 Analyzing the Doom code responsible for determining object placement

```

void P_XYMovement (mobj_t* mo) {
    int xmove;
    int ymove;

    // Cap movement in all directions
    if (mo->momx > MAXMOVE)
        mo->momx = MAXMOVE;
    else if (mo->momx < -MAXMOVE)
        mo->momx = -MAXMOVE;

    if (mo->momy > MAXMOVE)
        mo->momy = MAXMOVE;
    else if (mo->momy < -MAXMOVE)
        mo->momy = -MAXMOVE;

    xmove = mo->momx;
    ymove = mo->momy;

    do {
        // Divide fast movements into two steps
        if (xmove > MAXMOVE/2 || ymove > MAXMOVE/2) {
            ptrix = mo->x + xmove/2;
            ptriy = mo->y + ymove/2;
            xmove /= 2;
            ymove /= 2;
        }
        else {
            ptrix = mo->x + xmove;
            ptriy = mo->y + ymove;
            xmove = ymove = 0;
        }

        if (!P_TryMove (mo, ptrix, ptriy)) {
            // We collided with a wall, slide against it
            P_SlideMove (mo);
        }
    } while (xmove > 0 || ymove > 0);
}

else {
    if (gamekeydown[key_right]) // Move right
        cmd->angleturn -= angleturn[tspeed];
    if (gamekeydown[key_left]) // Move left
        cmd->angleturn += angleturn[tspeed];
}

if (gamekeydown[key_up]) // Move forward
    forward += forwardmove[speed];
if (gamekeydown[key_down]) // Move backward
    forward -= forwardmove[speed];

// (Section 2) Strafe right
if (gamekeydown[key_straferight])
    side += sidemove[speed];

// Strafe left
if (gamekeydown[key_strafeleft])
    side -= sidemove[speed];

// (Section 3) Cap speed
if (side > forwardmove[speed])
    side = forwardmove[speed];
else if (side < -forwardmove[speed])
    side = -forwardmove[speed];

cmd->forwardmove += forward;
cmd->sidemove += side;
}


```

against a wall where both collision checks will fail, `P_SlideMove` is called twice with equal momentum. Essentially, this means you are moving twice.

This can actually break the game quite well. In **Figure 4**, we see a level that contains a large amount of toxic blood. You can think of the level as a castle surrounded by a toxic moat. The red waste in this screenshot causes you a lot of damage, because it's supposed to keep you from skipping the castle that comprises most of the level. However, wallrunning allows the player to move so quickly that they can reach the other end with just enough health to survive to the exit. They essentially bypass the entire level as it was intended.

There is more strangeness in play here; it just so happens that if the player and the exit were reversed in this level, the bug would not trigger. That's because this bug only occurs when the player is moving in two of the four cardinal directions. Take a look at the code and see if you can spot why—again, it is all about that if statement in the loop. That if statement only checks to see if the momentum is greater than half of the maximum, so it will only call `P_TryMove` and `P_SlideMove` twice when moving quickly either north or east, since the other directions would be represented by a negative value. In the level shown in **Figure 4**, the bug worked because the player was moving north to the exit for the entire length of the toxic moat.

We could all see ourselves making this kind of mistake, and would probably never imagine that it would shatter world record times on certain levels. All you need to do to fix this is to either give `P_SlideMove` the correct momentum, or ensure that it is never called twice.

HEY, NOT TOO ROUGH We should not dwell on these failures; as you can see, it was some of the best programmers in the biz who made these mistakes. All of us are capable of writing code and only considering one case at a time (“Does moving forward work? Good. Does moving sideways work? Awesome.”), and never considering somebody will mash all of the keys at once. And on the bright side, sometimes, our bugs make a speedrunner’s day! 

Dave Wilkinson, commonly known as wilkie, is a systems researcher who seems to break more code than he writes. In his free time, he runs the open-source game coding competition (<http://osgcc.org>) to introduce game development to students.