

# DOOM-SAT

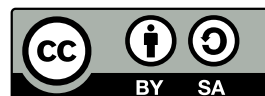
tms

July 14, 2025

## Abstract

3-SAT is implemented in the 1993 shooter *Doom* by id Software.

This work is licensed under a Creative Commons  
“Attribution-ShareAlike 4.0 International” license.



## 1 Introduction

In this story our hero faces the forces of H3ll. Instead of the doors and keycards normally used, H3ll has a far more sophisticated security system to keep interlopers out: a combination lock based on the boolean satisfiability problem.

This paper is organized into three parts: one giving a brief overview of 3-SAT, one giving a short primer on Doom maps and one going over the implementation of 3-SAT in Doom. Python code is also provided for generating boolean formulas given a list of integers.

For copyright reasons, illustrations in this paper are based on the freely licensed *Freedoom* IWADs[2]. For in-game screenshots the *Woof!* engine[4] was used, running *freedoom2.wad*.

## 2 3-SAT

The *boolean satisfiability problem* is the problem of determining, given a boolean formula, whether there exists an assignment of variables such that the formula evaluates to TRUE. 3-SAT is a specific variant of the problem where each clause in the formula has exactly three literals. The *conjunctive normal form* (CNF) is a conjunction (logical AND) of clauses, each of which consists of a disjunction (logical OR) of exactly three literals. A literal can either be a variable ( $x$ ) or the inversion of a variable ( $\neg x$ ). An example CNF formula is:

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee x_1 \vee \neg x_2)$$

This has the solutions  $x_1 = \text{TRUE}$  and  $x_2 = x_3 = \text{FALSE}$ . Note that the second clause has  $x_1$  twice, which is allowed.

The Cook–Levin theorem states that 3-SAT is NP-complete[1][5]. That is, that 3-SAT is NP and every NP problem can be transformed into 3-SAT. Therefore 3-SAT can serve as the basis for a good combination lock.

## 3 Doom maps

A Doom map is in essence a set of two-dimensional polygons, built up from four basic structures: vertices, linedefs, sidedefs and sectors.

### 3.1 Vertices

Vertices have an X position and a Y position. There can be a maximum of 32768 vertices in a Doom level.

### 3.2 Linedefs

Linedefs span from one vertex to another, and can have one or two sidedefs associated with it, depending on whether it's a one-sided or two-sided linedef. A one-sided linedef defines a wall with nothing behind it, while a two-sided linedef's passability depends on the object in question (player or monster), the height difference between both sides of the linedef and what flags are set for that linedef. A linedef that has a non-zero "action special" (sometimes called its "type") may be triggered by certain events, such as:

- The player crossing the linedef
- The player firing a hitscan weapon across the linedef
- The player pressing "use" on the linedef
- A monster crossing the linedef
- A missile crossing the linedef

Actions that can be triggered include things like:

- Opening/closing doors
- Raising/lowering platforms and lifts
- Turning on/off lights
- Teleporting players and/or monsters
- And much more

A linedef also has an integer called its "sector tag" associated with it, which decides which sector(s) are affected by the action when triggered. Sector tags are signed 16-bit integers which means tags can range from 0-32767. There can be a maximum of 65535 linedefs in a Doom level.

### 3.3 Sidedefs

A sidedef defines each "side" of a linedef and specifies the texture(s) used for that side of the linedef (upper, middle and lower texture) as well as the index (not tag!) of the sector that this sidedef faces. There can be a maximum of 32768 sidedefs in a Doom level.

### 3.4 Sectors

A sector is typically a single polygon. Technically a sector can be composed of multiple non-contiguous polygons, none of which necessarily must be closed. Each sector has a tag. Multiple sectors can use the same tag, which means a tag can be used to refer to multiple sectors. There can be a maximum of 32768 sectors in a Doom level.

## 4 3-SAT building blocks

### 4.1 Variables

A variable is a sector tag in this implementation.

Each variable has two buttons associated with it: one button to set it, and one button to clear it. The map is set up such that all variables are initialized to zero.

Each button is a linedef triggering the raising or lowering of all sector floors tagged by that linedef. Specifically, the action special "64 SR Floor To Lowest Adjacent Ceil" (LAC) is used to set a variable, and the action special "60 SR Floor To Lowest Adjacent Floor" (LAF) is used to clear a variable. Pressing a button thus causes all tagged floors to either rise to a level set by an adjacent sector coming "down" from the ceiling, or lowering the segments back to their original positions, which is always the same as the surrounding floor height. See figure 1.

Due to signed 16-bit integers being used for tags, and needing one tag for geometry that doesn't move (typically tagged zero), there can be up to 32767 variables in the original Doom. Other constraints in the map format, such as the maximum number of vertices and sectors, makes the number of variables lower in practice. We will get to this in section 5.1.

Yet another constraint is the number of so-called visplanes. We will not concern ourselves with this limit here, since maps can be designed with sufficient occlusion to make visplane concerns null.

### 4.2 Literals

A literal is defined by the passability through a section of the map. Multiple rules specify whether the player can pass through a two-sided linedef, but for this paper we only care about one: height difference.

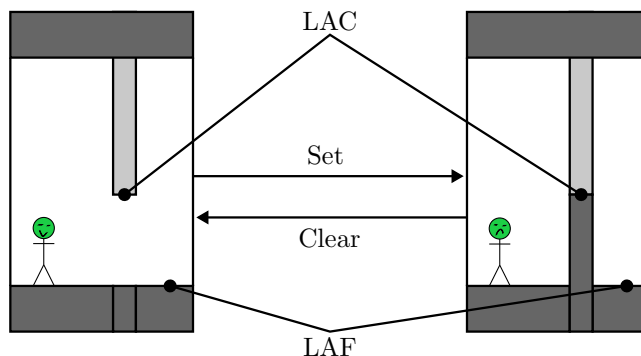


Figure 1: A variable (floor) being set/cleared. The direction of player movement is left to right. The light gray sector is in the background, to the side of the sector being raised/lowered. When set, the sector in front of the player forms a wall that blocks their way. This is the negative literal.

In the original Doom (and Doom II) released in 1993 the player can pass through a linedef if the sector floor on the other side is either lower than the current sector's floor, or no more than 24 units higher (assuming the player is standing on the floor). The latter rule is used to implement stairs. Some forks implement heretical features, most notably jumping, which we must take into account. The player can jump up to 36 units vertically using default settings in most forks. This means that a height difference of 61 or greater cannot be crossed in any fork the author is aware of, except by rocket jumping. In this work a height difference of 64 has been chosen for impassable walls.

Each literal consists of a "gate" structure and an "indicator" structure. The gate is what the player passes through. The indicators are placed such that the player can see them when operating the buttons, so that they don't have to run the "maze" every time they change its state to see if they can pass through. Indication is also implemented via the raising and lowering of floors, using the same sector tag as the corresponding literal.

#### 4.2.1 Negative literals

A negative literal means the section can be passed in its initial state. The realization is shown in figures 1, 2 and 3. When the "set" button is pressed, a floor segment is raised 64 units, blocking the player's path. The height of 64 is set by the small square to the right of the gate in figure 2, by having its ceiling be at the desired height.

The negative indicator is visible by default, and covered when the player pushes the "set" button. This is visible in the foreground in figure 3.

The negative literal uses 6 vertices, 7 linedefs and 2 sectors for the gate, and 8 vertices, 8 linedefs and 2 sectors for the indicator. The number of sidedefs depends on texture choice and sidedef compression, and can be as low as 2 while

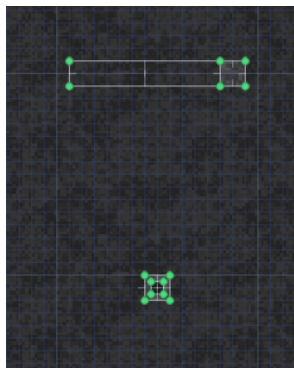


Figure 2: Negative literal in Slade[3]. Gate on top, indicator on the bottom.

maintaining indicator functionality.

#### 4.2.2 Positive literals

A positive literal means the section cannot be passed unless the "set" button has been pressed. This means its default state should be an impassable 64 unit high wall. The idea to make the wall passable is to raise a small set of stairs. The number of steps raised must be more than one, because a single step cannot bring the player up more than 24 units in *Doom*, leaving 40 more units to scale. Two steps will do the job. Wrapping the steps around the wall is useful for being able to backtrack. The realization is shown in figures 4 and 5. Figure 6 shows a schematic view.

The positive indicator is not visible by default. It is raised when the player pushes the "set" button.

The positive literal uses 15 vertices, 19 linedefs and 5 sectors for the gate, and 8 vertices, 8 linedefs and 2 sectors for the indicator. The number of sidedefs can again be as low as 2.

### 4.3 Clauses

A clause consists of three literals. More or fewer literals could also be used, but for this work three literals are always used. The three literals are placed next to each other in such a way that either evaluating to TRUE allows the player to pass through that clause. Multiple clauses can be chained together in many ways. For this work an horizontal wave-like arrangement is used, as illustrated in figure 7. Each short horizontal line corresponds to a literal, and each clause is a group of three such literals. Walls are placed such that the player cannot jump from say the top of a positive literal to the top of a negative in the next clause, which might be possible if the clauses were arranged in a linear north-south fashion. The player is thus forced to pass through the structure in the intended way.

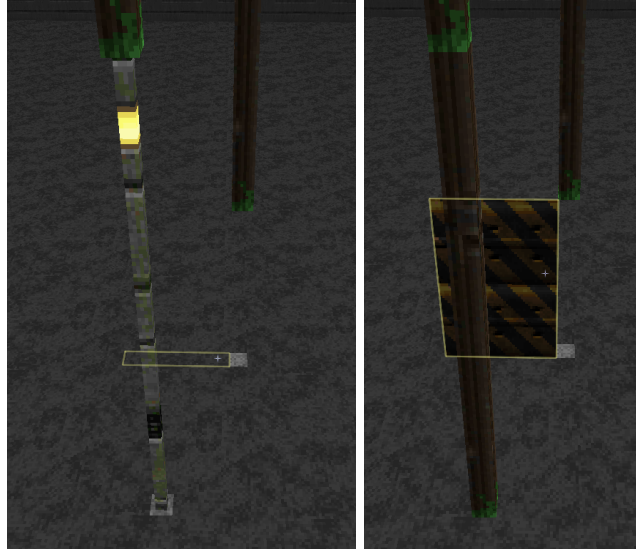


Figure 3: Negative literal in 3D in Slade. Cleared and passable state ( $\neg\text{FALSE} = \text{TRUE}$ ) on the left, set and impassable state ( $\neg\text{TRUE} = \text{FALSE}$ ) on the right. Note the visible indicator on the left, and its occluded state on the right.

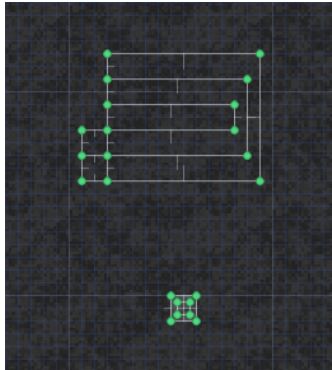


Figure 4: Positive literal in Slade. Gate on top, indicator on the bottom. The two small squares on the left set the height of each "step".

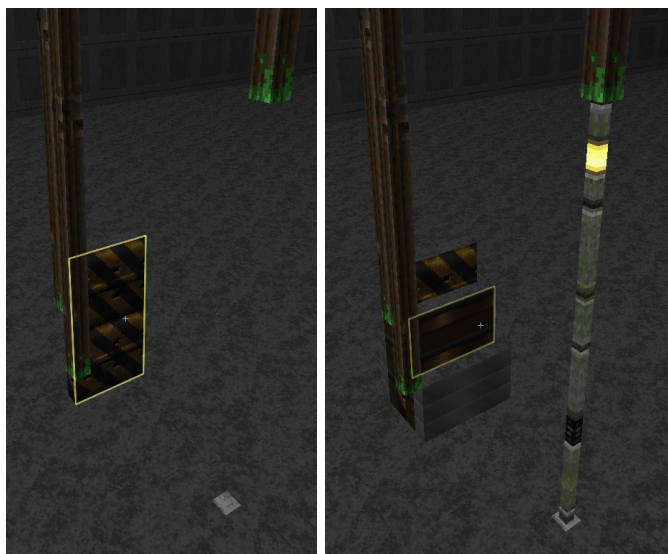


Figure 5: Positive literal in 3D in Slade. Cleared and impassable state on the left, set and passable state on the right. Note the invisible indicator on the left, and the raised and visible indicator on the right.

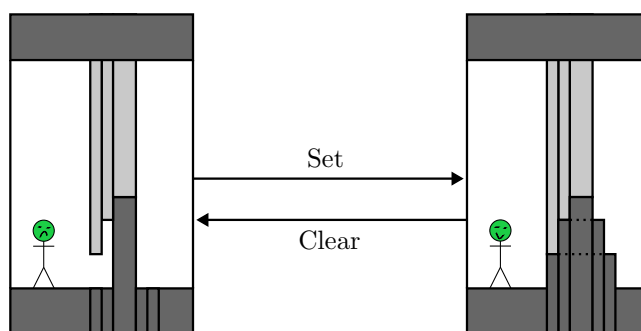


Figure 6: Positive version of 1. As before, the light gray sectors are to the side. When the variable is set, stairsteps are raised, making the literal passable.

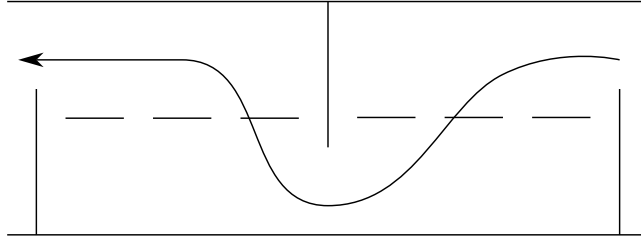


Figure 7: Clauses arranged in a horizontal manner such that the player passes from right to left in a wave-y manner.

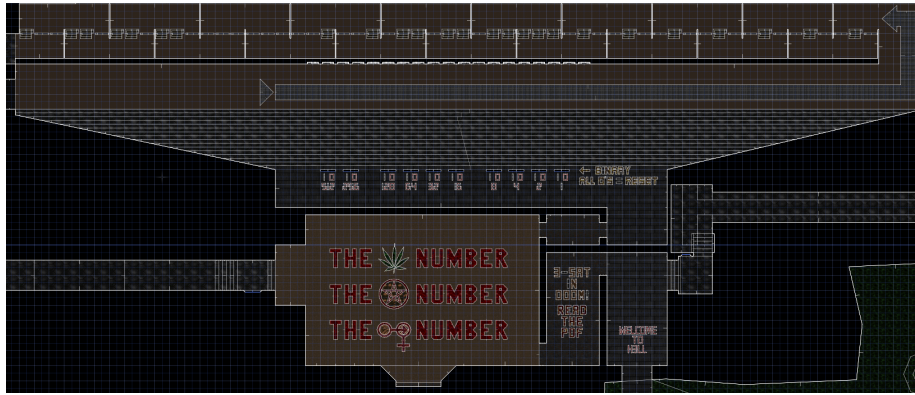


Figure 8: Full implementation in Slade.

## 5 Implementation

The full implementation is shown in figures 8 and 9. It has 19 clauses and 10 variables. The player enters from the southeast. The buttons are on the platform overlooking the entire structure, accompanied by the powers of two between 1 and 512. The indicators look different from those in figures 3 and 5 for user experience (UX) reasons. A shorter distance between floor and ceiling resulted in faster indicator updates. Making the indicators wider made them more visible. Each group of three indicators is flanked by a dark texture to make the grouping more obvious. The three indicators for each clause takes up 64 map units in width, whereas the clause itself takes up 192 units. The narrower dimensions of the indicators makes it possible for the player to see them all. The clauses themselves are behind this indicator "front panel". Figure 10 gives an in-game view of the clauses themselves.

A program listing is given in appendix A that, given the number of variables  $nbits$  and a list of integers  $ints$ , finds a set of clauses that, when the variables are interpreted as an integer, evaluates to TRUE for the integers in  $ints$  and no others. It can therefore be used to implement a combination lock with more





Figure 9: Full implementation in-game. The set button for bit 9 is active and indicators in clauses 2, 3, 9 and 18 are moving (blue arrows). Clauses 3 and 18 (orange lines) are about to evaluate to FALSE. Clause 11 (red line) is already FALSE (indicator window completely black).



Figure 10: In-game view between clause 10 (right) and clause 11 (left) in the initial all-zeroes configuration. Clause 11 is impassable.

than one solution. The program runs indefinitely, outputting the shortest list of clauses found so far, with negative integers corresponding to negative literals, and positive integers to positive literals. For some sets of integers the shortest list of clauses isn't always the most interesting, especially when *do\_random* is set to *False*.

## 5.1 Limitations

As mentioned earlier in this paper, the number of variables is limited by limitations in the Doom map format. The table below summarizes how the map format limits the number of variables for each type of literal.

Literal	Vertices	Linedefs	Sectors
Negative	2340	4369	8191
Positive	1424	2427	4681

The above values were computed by dividing the limit on the number of vertices, linedefs and sectors by the number of such elements required by each literal. Sidedefs are not considered since they are easily compressed. We see that the lowest value is  $\lfloor 32768/(15+8) \rfloor = 1424$ . In other words, that vertices are the limiting factor. If we can't control the number of positive literals then we must take this as the upper bound to the number of literals in this implementation. In practice the number of literals is even lower due to the need for buttons and other geometry.

## 6 Conclusions

3-SAT has been implemented in Doom, with satisfiability defined as physical passability through in-game space. A program has been written for generating 3-SAT formulas which evaluate to TRUE for only a given list of integers and no others.

## 7 Future work

It may be possible to implement non-radix-2 literals by using lifts that stop on multiple levels. The author suspects the necessary added geometry for doing this make it not worthwhile, but perhaps there is a "sweet spot" for ternary or quaternary logic.

A more economical set of literal designs could increase the number of possible clauses while keeping within Doom's limits.

Finally, the present work could be adapted for real-world binary combination locks that are effectively impossible to "feel" one's way to the combination for, unlike conventional combination locks.

## References

- [1] Stephen Cook. The complexity of theorem proving procedures. *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, page 151–158, 1971.
- [2] Freedom developers. Freedom. <https://freedom.github.io/>, 2024. Accessed 2024-02-06.
- [3] Slade developers. Slade. <https://slade.mancubus.net/>, 2024. Accessed 2024-03-06.
- [4] Woof! developers. Woof! <https://github.com/fabiangreffrath/woof>, 2024. Accessed 2024-02-06.
- [5] Leonid Levin. Универсальные задачи перебора (universal search problems). *Problems of Information Transmission*, 9:115–116, 1973.

## A doom\_sat.py

```
from random import randint, seed

def run(nbits, ints, do_random):
    solutions = list(range(1 << nbits))

    clause = [-1, 0, 0]
    clauses = []

    while len(solutions) > len(ints):
        #print(solutions)
        if do_random:
            clause = (
                randint(0, nbits-1),
                randint(0, nbits-1),
                randint(0, nbits-1)
            )
        else:
            clause[0] += 1
            if clause[0] >= nbits:
                clause[0] = 0
            clause[1] += 1
            if clause[1] >= nbits:
                clause[1] = 0
            clause[2] += 1
            if clause[2] >= nbits:
                print('tried all, failing')
                exit(1)

        # Try all eight sign combinations,
        # or a random one if we're randomizing
        signs = [randint(0,7)] if do_random else range(8)
        for sign in signs:
            s0 = sign & 1
            s1 = (sign >> 1) & 1
            s2 = (sign >> 2) & 1

            is_ok = True
            for i in ints:
                o0 = ((i >> clause[0]) & 1) ^ s0
                o1 = ((i >> clause[1]) & 1) ^ s1
                o2 = ((i >> clause[2]) & 1) ^ s2

                if o0 == 0 and o1 == 0 and o2 == 0:
```

```

        is_ok = False
        break

    if is_ok:
        # Check if this clause removes any undesired solutions
        new_sols = []
        for i in solutions:
            o0 = ((i >> clause[0]) & 1) ^ s0
            o1 = ((i >> clause[1]) & 1) ^ s1
            o2 = ((i >> clause[2]) & 1) ^ s2
            if o0 == 1 or o1 == 1 or o2 == 1:
                new_sols.append(i)

        if len(new_sols) < len(solutions):
            #print(f'{len(solutions)} -> {len(new_sols)} solutions')
            solutions = new_sols
            # Need to add 1 because 0 == -0
            clauses.append((
                (1 + clause[0]) * (1 - 2 * s0),
                (1 + clause[1]) * (1 - 2 * s1),
                (1 + clause[2]) * (1 - 2 * s2)
            ))

    return clauses

#seed(0)
nbits = 10
ints = [123, 321]
do_random = True
best = None

while True:
    clauses = run(nbits, ints, do_random)
    if best is None or len(clauses) < len(best):
        best = clauses
        print(clauses)
        print(f'{len(best)} clauses')

```