

Tetris NES to SRS

A report compiled by Chase Dillard and Atreyu McLewin
github.com/GingerDeity/TetrisNES

Introduction

Tetris is one of those classic games everyone knows about, and the version built for the Nintendo Entertainment System (NES) is one of the most famous renditions of that classic! However, this game was created in 1989, just two years before the Standard Rotation System (a system nearly all subsequent Tetris games followed) was created. As a result of this, the NES version used its own system that is significantly different from its successors. In this project, we intended to add in as many aspects of the Standard Rotation System (SRS) as much as possible. This revolved around two major features: adding in missing piece rotations, and adding in wall-kicking!

Our approach for this project was twofold in directions. First, we want to modify as little of the existing logic as possible, as NES development is already difficult enough. Ideally any changes we made would be relatively unintrusive, and only affect data instead of functions. Another key approach of the project was using three different environments, each with their own purpose. One environment would be used for developing the new ROM, another for translating and commenting the original ROM, and another for playtesting our ROMs. We used VSCode, Ghidra, and Mesen emulator for each environment respectively.

Chase was primarily responsible for implementing wallkicks and early translation of decompiled code, while Atreyu implemented the new piece rotations and set up the development environments. With all this in mind, let's get started!

General Challenges & Solutions

Before delving into our development journey, it's important to first note some problems regarding NES memory that had lasting repercussions on our NES development. One problem was that modifying NES memory is very difficult, and even getting started with development was tricky due to the original ROM itself needing a special configuration file that we made ourselves. What this configuration file did was specify another 8KB ROM memory bank, dubbed ROM2. This was used to specify where a TILES segment could be located, something that was very necessary for drawing sprites but was somehow not being included in original builds of the ROM.

Screenshots of added memory banks in my-nes.cfg

```
MEMORY {
    ZP:      file = "", start = $0002, size = $001A, type = rw, define = yes;

    # INES Cartridge Header
    HEADER:  file = %, start = $0000, size = $0010, fill = yes;

    # 2 16K ROM Banks
    # - startup
    # - code
    # - rodata
    # - data (load)
    ROM0:    file = %, start = $8000, size = $7FFA, fill = yes, define = yes;

    # Hardware Vectors at End of 2nd 8K ROM
    ROMV:    file = %, start = $FFFA, size = $0006, fill = yes;

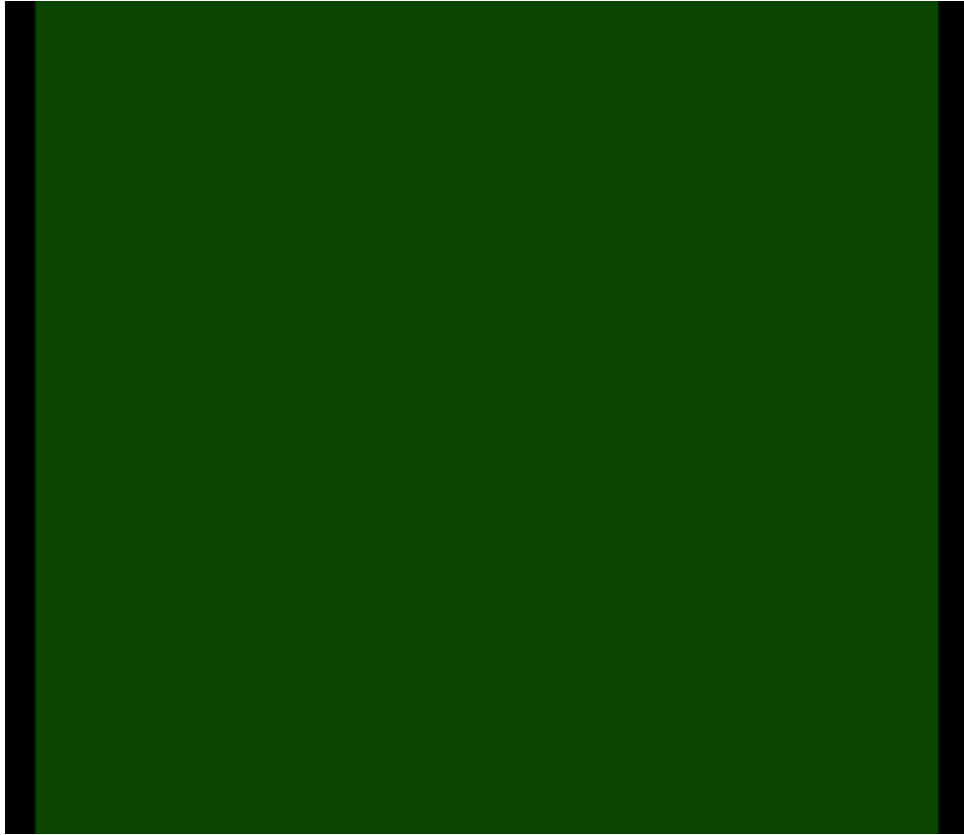
    # 1 8k CHR Bank
    ROM2:    file = %, start = $0000, size = $4000, fill = yes;

SEGMENTS {
    ZEROPAGE: load = ZP,          type = zp;
    HEADER:   load = HEADER,      type = ro;
    LOWCODE:  load = ROM0,        type = ro, optional = yes;
    ONCE:     load = ROM0,        type = ro, optional = yes;
    CODE:     load = ROM0,        type = ro, define = yes;
    RODATA:   load = ROM0,        type = ro, define = yes;
    DATA:    load = ROM0, run = RAM, type = rw, define = yes;
    VECTORS:  load = ROMV,        type = rw;
    BSS:      load = RAM,         type = bss, define = yes;

    # ADDED - 3/31, 6:32p
    TILES:    load = ROM2,        type = rw;
}
```

However, adding new instructions and memory to the NES ROM is still difficult, because simply adding new things in the middle of ROM0 data (where all the code is held) is enough to cause significant misalignments, resulting in graphical errors.

Screen of Tetris with misaligned data



To prevent this, our general solution was to add our new data and functions to the end of the ROMo segment (which contained padding bytes that could be easily removed) and simply have the existing code reference our tables and functions instead. If we needed to add to what a function is doing, we would replace the first few instructions with a jump to a custom function at the end of ROMo, and deal with any misalignments with small but never-used instructions after that jump.

Before (top) & After (bottom) example of adding new data

```
.byte $00, $00, $00, $ff, $bf, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff ; $FEBD
.byte $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff ; $FECD
.byte $ff, $ff, $ff, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00 ; $FEDD
.byte $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00 ; $FEED
.byte $00, $00, $00 ; $FEFD

Reset:
    cld ; $FF00 D8
    sei ; $FF01 78
    ldx #$00 ; $FF02 A2 00
    stx PPU_CTRL ; $FF04 8E 00 20
    stx PPU_MASK ; $FF07 8E 01 20
```

```

.byte $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00 ; $FEAD
.byte $00, $00, $00, $ff, $bf, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ef, $ff, $ff ; $FEBD
.byte $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff, $ff ; $FECD
.byte $ff, $ff, $ff, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00

_tetrimino_types:
.byte $00, $00, $00, $00, $01, $01, $01, $01, $02, $02, $02, $02, $03, $03, $03, $03
.byte $04, $04, $04, $04, $05, $05, $05, $05, $06

Reset:
    cld                ; $FF00 D8
    sei                ; $FF01 78
    ldx #$00           ; $FF02 A2 00
    stx PPU_CTRL       ; $FF04 8E 00 20
    stx PPU_MASK       ; $FF07 8E 01 20

```

Example of adding jumps, 88AB points to our function now

```

1071 rotate_tetrimino:
1072     jmp rotate_tetrimino_new      ; $88AB A5 42
1073     clc                          ; $88AD 85 AE
1074     clc                          ; $88AF 18
1075     lda z:current_piece          ; $88B0 A5 42
1076     asl a                        ; $88B2 0A
1077     tax                          ; $88B3 AA

7024 rotate_tetrimino_new:
7025     lda z:tetriminoX
7026     sta copyTetriminoX
7027     lda z:tetriminoY
7028     sta copyTetriminoY
7029     lda z:current_piece
7030     sta copyCurrentPiece

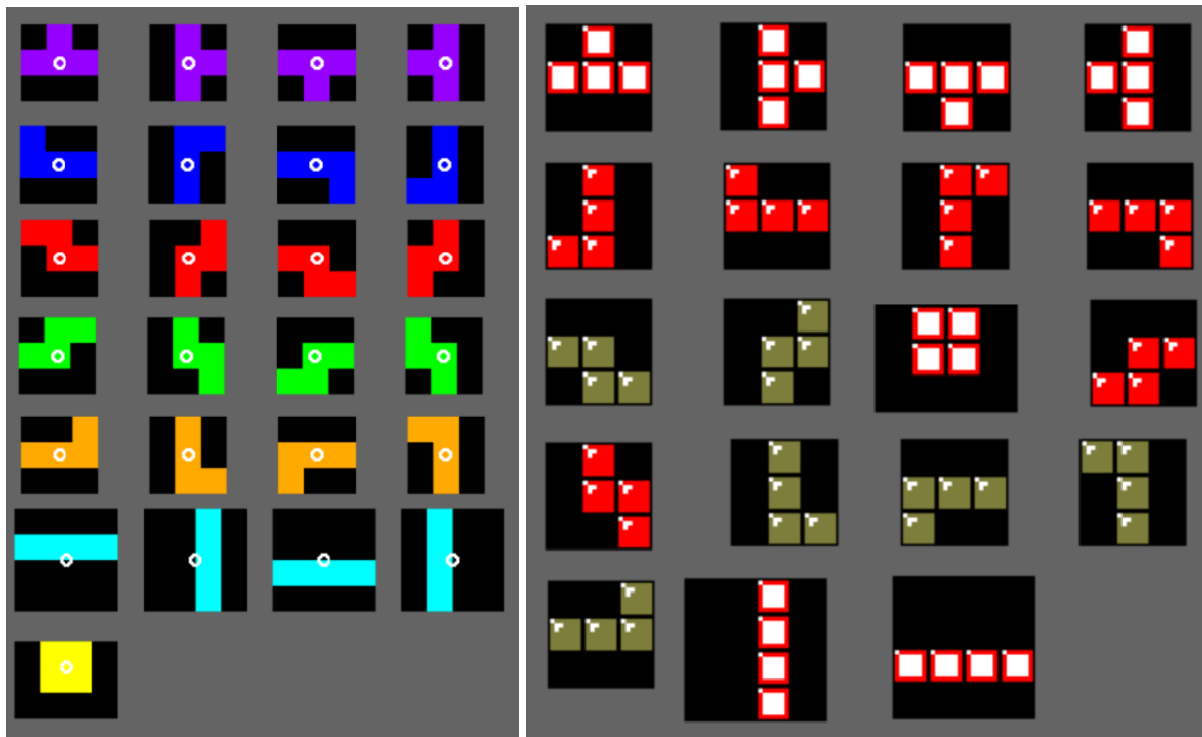
```

With these limitations in mind, let's finally dive into what we added and how, starting with filling in the rotation states!

Rotation States

In Tetris, there are 7 different *tetrominoes* (we switch between calling them *pieces* and *tetrominoes* throughout this report), and each tetromino has 4 *blocks* each. In the following diagram, the left picture shows the *T*, *J*, *Z*, *S*, *L*, *I*, and *O* tetrominoes from top to bottom. In the original Tetris ROM, there are only 19 possible *rotation states* between all 7 pieces, each represented by a byte in the code that we refer to as *rotation values* or *piece IDs*.

Rotation States in SRS (left) vs NES (right)



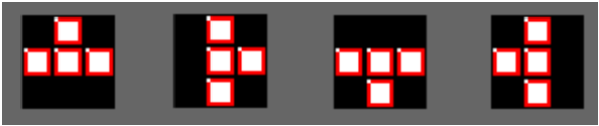
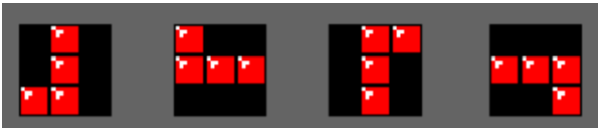


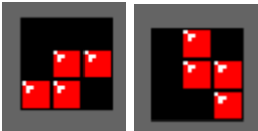
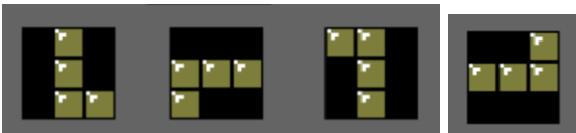
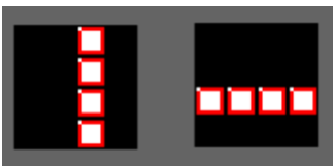
As you can see, Tetris NES is missing 6 rotation states that are present in future games. There were 2 missing rotations for the I, Z, and S pieces each. These tetrominos, which are rotationally symmetric, did not have both left/right and both up/down states that the piece could orient itself in! There are three steps to adding in these missing states to the game: telling the game that there are new states, drawing them, and being able to spawn them.

Rotation State Table (Original Tetris ROM)

```
old_rotation_table:
.byte $03, $01, $00, $02, $01, $03, $02, $00, $07, $05, $04, $06, $05, $07, $06, $04 ; $88EE
.byte $09, $09, $08, $08, $0a, $0a, $0c, $0c, $0b, $0b, $10, $0e, $0d, $0f, $0e, $10 ; $88FE
.byte $0f, $0d, $12, $12, $11, $11 ; $890E
```

The above table, located at 0x88EE, holds all the possible states for pieces in the game. Each rotation has 2 entries in the table, denoting either a clockwise (CW) or counterclockwise (CCW) rotation. In a function we labelled 'rotate_tetromino' located at 0x88AB, the game checks if a player is pressing either the A or B button and pulls one of the states out of this table accordingly! The current rotation state of the piece is used as an index into this table. The following table shows the rotation values for each of the pieces in the original ROM.

Rotation Values (Original Tetris ROM)

Rotation Value	Associated Rotation
0x0, 0x1, 0x2, 0x3	
0x4, 0x5, 0x6, 0x7	
0x8, 0x9	
0xA	
0xB, 0xC	
0xD, 0xE, 0xF, 0x10	
0x11, 0x12	

The *old_rotation_table* shown earlier holds the rotation states to go to for each rotation value in order, starting with the CCW rotation and then the CW rotation. For example, the first 2 entries in the table are T-up CCW and T-up CW.

Rotation Table Entries for T-Tetromino


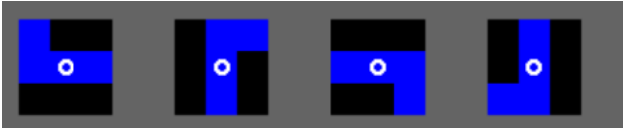





```

;               ===T-BLOCK===
; i:            0,  1,  2,  3,  4,  5,  6,  7,
; curr_piece: $00, $00, $01, $01, $02, $02, $03, $03
; rotation:   CCW CW  CCW CW  CCW CW  CCW CW
; rt[i]:      $03, $01, $00, $02, $01, $03, $02, $00

```

Our challenge was to add in a way to access new states to the game, which led to the new rotation values shown below. Modifying the rotation table to accommodate for these new states increased its size from 38 bytes to 50 bytes. It's worth noting that while the order of tetrominoes (*T*, then *J*, etc) was kept largely intact, we did move the O-piece to the end of the table.

Rotation Values (Modified Tetris ROM)

Rotation Value	Associated Rotation
0x0, 0x1, 0x2, 0x3	
0x4, 0x5, 0x6, 0x7	
0x8, 0x9, 0xa, 0xB	
0xC, 0xD, 0xE, 0xF	
0x10, 0x11, 0x12, 0x13	
0x14, 0x15, 0x16, x17	
0x18	

Drawing the Rotations

While the game knows that, in principle, there are indeed now 25 rotation states instead of just 19, it still has no idea how to draw those new pieces. To fix this, we have to understand another table located at 0x8A9C:

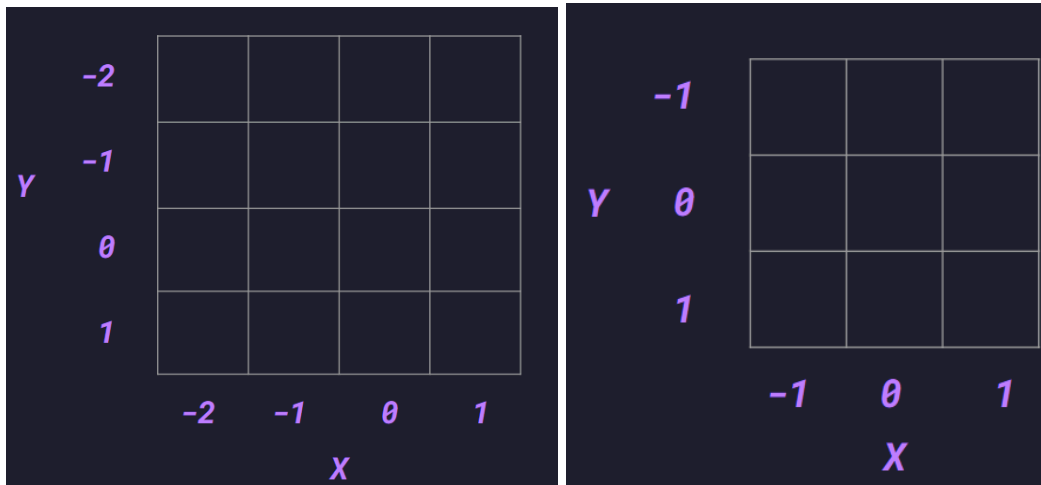
Graphics Table (Original Tetris ROM)

```
old_orientation_table:
.byte $00, $7b, $ff, $00, $7b, $00, $00, $7b, $01, $ff, $7b, $00, $ff, $7b, $00, $00 ; $8A9C
.byte $7b, $00, $00, $7b, $01, $01, $7b, $00, $00, $7b, $ff, $00, $7b, $00, $00, $7b ; $8AAC
.byte $01, $01, $7b, $00, $ff, $7b, $00, $00, $7b, $ff, $00, $7b, $00, $01, $7b, $00 ; $8ABC
.byte $ff, $7d, $00, $00, $7d, $00, $01, $7d, $ff, $01, $7d, $00, $ff, $7d, $ff, $00 ; $8ACC
.byte $7d, $ff, $00, $7d, $00, $00, $7d, $01, $ff, $7d, $00, $ff, $7d, $01, $00, $7d ; $8ADC
.byte $00, $01, $7d, $00, $00, $7d, $ff, $00, $7d, $00, $00, $7d, $01, $01, $7d, $01 ; $8AEC
.byte $00, $7c, $ff, $00, $7c, $00, $01, $7c, $00, $01, $7c, $01, $ff, $7c, $01, $00 ; $8AFC
.byte $7c, $00, $00, $7c, $01, $01, $7c, $00, $00, $7b, $ff, $00, $7b, $00, $01, $7b ; $8B0C
.byte $ff, $01, $7b, $00, $00, $7d, $00, $00, $7d, $01, $01, $7d, $ff, $01, $7d, $00 ; $8B1C
.byte $ff, $7d, $00, $00, $7d, $00, $00, $7d, $01, $01, $7d, $01, $ff, $7c, $00, $00 ; $8B2C
.byte $7c, $00, $01, $7c, $00, $01, $7c, $01, $00, $7c, $ff, $00, $7c, $00, $00, $7c ; $8B3C
.byte $01, $01, $7c, $ff, $ff, $7c, $ff, $ff, $7c, $00, $00, $7c, $00, $01, $7c, $00 ; $8B4C
.byte $ff, $7c, $01, $00, $7c, $ff, $00, $7c, $00, $00, $7c, $01, $fe, $7b, $00, $ff ; $8B5C
.byte $7b, $00, $00, $7b, $00, $01, $7b, $00, $00, $7b, $fe, $00, $7b, $ff, $00, $7b ; $8B6C
.byte $00, $00, $7b, $01, $00, $ff, $00, $00, $ff, $00, $00, $ff, $00, $00, $ff, $00 ; $8B7C
.byte $a5, $a2, $0a, $0a, $85, $a8, $0a, $18, $65, $a8, $a8, $a6, $b3, $a9, $04, $85 ; $8B8C
.byte $a9, $b9, $9c, $8a, $18, $0a, $0a, $0a, $65, $a1, $9d, $00, $02, $e8, $c8, $b9 ; $8B9C
.byte $9c, $8a, $9d, $00, $02, $e8, $c8, $a9, $02, $9d, $00, $02, $e8, $b9, $9c, $8a ; $8BAC
.byte $18, $0a, $0a, $0a, $65, $a0, $9d, $00, $02, $e8, $c8, $c6, $a9, $d0, $d2, $86 ; $8BBC
.byte $b3, $60 ; $8BCC
```

This is the table that's responsible for drawing the tetromino sprites in the game. While it looks scary at first, the structure is surprisingly simple once you understand it. This table contains 4 sets of bytes (one for each block in a piece) for every piece in the game. Each set contains three bytes in the following order: a Y position, a sprite ID, and an X position. After the first 228 bytes in the table, there's some padding bytes followed by low-level function code that actually draws the sprites to the screen.

If you look closely, you'll see that all the X and Y positions are either -2, -1, 0, or 1 (in 2's complement). This is because these aren't X and Y positions for the screen, but rather for a given tetromino's sprite bounding box. The below tables shows how the game translates these X and Y positions to a grid, and an example of how a T-tetromino is drawn.

Sprite Bounding Boxes



Drawing the T-tetromino



Once we understood the table, adding in the new data was fairly simple (though still rather tedious and difficult to check, leading to python scripts that showed us what our bytes were doing). Adding in the new entries resulted in a size increase from 306 bytes to 378 bytes. The new table is shown below:

Graphics Table (Modified Tetris ROM)

```
orientation_table:
.byte $00, $7b, $ff, $00, $7b, $00, $00, $7b, $01, $ff, $7b, $00, $ff, $7b, $00, $00 ; T-BLOCK
.byte $7b, $00, $00, $7b, $01, $01, $7b, $00, $00, $7b, $ff, $00, $7b, $00, $00, $7b
.byte $01, $01, $7b, $00, $ff, $7b, $00, $00, $7b, $00, $00, $7b, $ff, $01, $7b, $00
.byte $00, $7d, $ff, $00, $7d, $00, $00, $7d, $01, $ff, $7d, $ff, $ff, $7d, $00, $00 ; J-BLOCK
.byte $7d, $00, $01, $7d, $00, $ff, $7d, $01, $00, $7d, $ff, $00, $7d, $00, $00, $7d
.byte $01, $01, $7d, $01, $ff, $7d, $00, $00, $7d, $00, $01, $7d, $00, $01, $7d, $ff
.byte $ff, $7c, $ff, $00, $7c, $00, $00, $7c, $01, $ff, $7c, $00, $ff, $7c, $01, $00 ; Z-BLOCK
.byte $7c, $00, $00, $7c, $01, $01, $7c, $00, $00, $7c, $ff, $00, $7c, $00, $01, $7c
.byte $00, $01, $7c, $01, $01, $7c, $ff, $00, $7c, $00, $00, $7c, $ff, $ff, $7c, $00
.byte $00, $7d, $ff, $00, $7d, $00, $ff, $7d, $00, $ff, $7d, $01, $ff, $7d, $00, $00 ; S-BLOCK
.byte $7d, $00, $00, $7d, $01, $01, $7d, $01, $01, $7d, $ff, $00, $7d, $00, $01, $7d
.byte $00, $00, $7d, $01, $ff, $7d, $ff, $00, $7d, $00, $00, $7d, $ff, $01, $7d, $00
.byte $ff, $7c, $01, $00, $7c, $00, $00, $7c, $ff, $00, $7c, $01, $ff, $7c, $00, $00 ; L-BLOCK
.byte $7c, $00, $01, $7c, $00, $01, $7c, $01, $00, $7c, $ff, $00, $7c, $00, $00, $7c
.byte $01, $01, $7c, $ff, $ff, $7c, $ff, $00, $7c, $00, $ff, $7c, $00, $01, $7c, $00
.byte $ff, $7b, $fe, $ff, $7b, $ff, $ff, $7b, $00, $ff, $7b, $01, $fe, $7b, $00, $ff ; I-BLOCK
.byte $7b, $00, $00, $7b, $00, $01, $7b, $00, $00, $7b, $fe, $00, $7b, $ff, $00, $7b
.byte $00, $00, $7b, $01, $fe, $7b, $ff, $ff, $7b, $ff, $00, $7b, $ff, $01, $7b, $ff
.byte $00, $7b, $ff, $00, $7b, $00, $01, $7b, $ff, $01, $7b, $00, $00, $ff, $00, $00 ; O-BLOCK (minus last 4 bytes)
.byte $ff, $00, $00, $ff, $00, $00, $ff, $00, $a5, $a2, $0a, $0a, $85, $a8, $0a, $18 ; PADDING + DISPLAY FUNC
.byte $65, $a8, $a8, $a6, $b3, $a9, $04, $85, $a9, $b9, $9c, $8a, $18, $0a, $0a, $0a
.byte $65, $a1, $9d, $00, $02, $e8, $c8, $b9, $9c, $8a, $9d, $00, $02, $e8, $c8, $a9
.byte $02, $9d, $00, $02, $e8, $b9, $9c, $8a, $18, $0a, $0a, $65, $a0, $9d, $00
.byte $02, $e8, $c8, $c6, $a9, $d0, $d2, $86, $b3, $60
```

Spawning New Blocks

While the game now knows about the new rotation states, and how to actually draw them, we still need a way to spawn in the new tetriminos. We were able to learn much about how the game does this, and implement it partially, but there were certain parts of this endeavor that were not successful. For now, let's start with how the game determines what pieces to spawn. There are 4 tables that are crucial for the game to know how to spawn blocks, which we've labelled 'tetronimo_types', 'spawn_rotate', 'next_to_curr', and 'nextIDtoSprite.'

Tetromino Types Table (Original Tetris ROM)

```
old_tetromino_types:
.byte $00, $00, $00, $00, $01, $01, $01, $01, $02, $02, $03, $04, $04, $05, $05, $05 ; $993B
.byte $05, $06, $06 ; $994B
```

Spawn Rotation Table (Original Tetris ROM)

```
_old_spawn_rotate:
.byte $02, $07, $08, $0a, $0b, $0e, $12, $02 ; $994E
```

Next Piece ID to Current Piece ID (Original Tetris ROM)

```
_old_next_to_curr:
.byte $02, $02, $02, $02, $07, $07, $07, $07, $08, $08, $0a, $0b, $0b, $0e, $0e, $0e ; $9956
.byte $0e, $12, $12 ; $9966
```

Next Piece ID to Sprite ID (Original Tetris ROM)

```
_old_nextIDtoSprite:
.byte $00, $00, $06, $00, $00, $00, $00, $09, $08, $00, $0b, $07, $00, $00, $0a, $00 ; $8BE5
.byte $00, $00, $0c, $00, $00, $0f, $00, $00, $00, $00, $12, $11, $00, $14, $10, $00 ; $8BF5
.byte $00, $13, $00, $00, $00, $15, $00, $ff, $fe, $fd, $fc, $fd, $fe, $ff, $00, $01 ; $8C05
.byte $02, $03, $04, $05, $06, $07, $08, $09, $0a, $0b, $0c, $0d, $0e, $0f, $10, $11 ; $8C15
.byte $12, $13 ; $8C25
```

The `_tetromino_types` table simply divided the original 19 rotations into the 7 tetrominos. You can see the first 4 bytes correlate to the T-tetromino, the next 4 map to the J-tetromino, the next 2 are for the Z-tetromino, and so on. The `_spawn_rotate` table tells the game what the rotation state for the spawned piece should be, and you can see that it's indexed based on the 7 tetrominos, with the T-tetromino (first 4 entries) always being spawned at rotation state 0x2. This is because the original Tetris ROM won't spawn in just any tetromino, but instead the ones with the lowest Y-values. If you correlate the spawn rotations here to the original rotation values, you can see it clearly.

The `_next_to_curr` table maps all piece IDs into the IDs of pieces that we'd want to spawn, and the `_nextIDtoSprite` table holds sprite values at specific indexes (indexes that correlate to the rotation states the game would consider to be valid spawn rotations). While it's unclear to us why the developers thought they needed all four of these tables (as some like the `_next_to_curr` table could seemingly be replicated by just having the next piece ID map to the `_tetrimino_types` table and then mapping that value to `_spawn_rotate`), modifying all four of these tables were necessary both for spawning the correct pieces and having the correct piece displayed in the "NEXT" window in-game (a problem that was remedied upon discovery of the `_nextIDtoSprite` table). The new tables are shown below and are fairly easy to understand:

Tetromino Types Table (Modified Tetris ROM)

```
_tetrimino_types:
.byte $00, $00, $00, $00, $01, $01, $01, $01, $02, $02, $02, $02, $03, $03, $03, $03
.byte $04, $04, $04, $04, $05, $05, $05, $05, $06
```

Spawn Rotation Table (Modified Tetris ROM)

```
_spawn_rotate:
.byte $02, $06, $0a, $0e, $12, $16, $18, $02
```

Next Piece ID to Current Piece ID (Modified Tetris ROM)

```
_next_to_curr: ; Translates the next piece's ID to the curr piece ID
.byte $02, $02, $02, $02, $06, $06, $06, $06, $0a, $0a, $0a, $0a, $0e, $0e, $0e, $0e ; T, J, Z, S BLOCKS
.byte $12, $12, $12, $12, $16, $16, $16, $16, $18, $18; L, I, O BLOCKS
```

Next Piece ID to Sprite ID (Modified Tetris ROM)

```
_nextIDtoSprite:
.byte $00, $00, $06, $00, $00, $00, $09, $00, $00, $00, $08, $00, $00, $00, $07, $00
.byte $00, $00, $0a, $00, $00, $00, $0c, $00, $0b, $00, $00, $0f, $00, $00, $00, $00
.byte $12, $11, $00, $14, $10, $00, $00, $13, $00, $00, $00, $15, $00, $ff, $fe, $fd
.byte $fc, $fd, $fe, $ff, $00, $01, $02, $03, $04, $05, $06, $07, $08, $09, $0a, $0b
.byte $0c, $0d, $0e, $0f, $10, $11, $12, $13, $14, $15, $16, $17, $18, $19
```

So, that should be it, right? We now have a Tetris ROM-hack with 6 new rotation states and the means to both draw them and spawn them in! Well unfortunately, there was one oversight that was so critical it deserves its own section.

Orientation Overflows

There's one other very crucial function that we haven't discussed yet, which we labelled as *is_position_valid()*. What it essentially does is check every block in a tetromino pieces' condition every time that tetromino's position is altered, whether through being spawned in, rotated, moved downwards, or hitting a border or other piece. The disassembled Ghidra code is shown below.

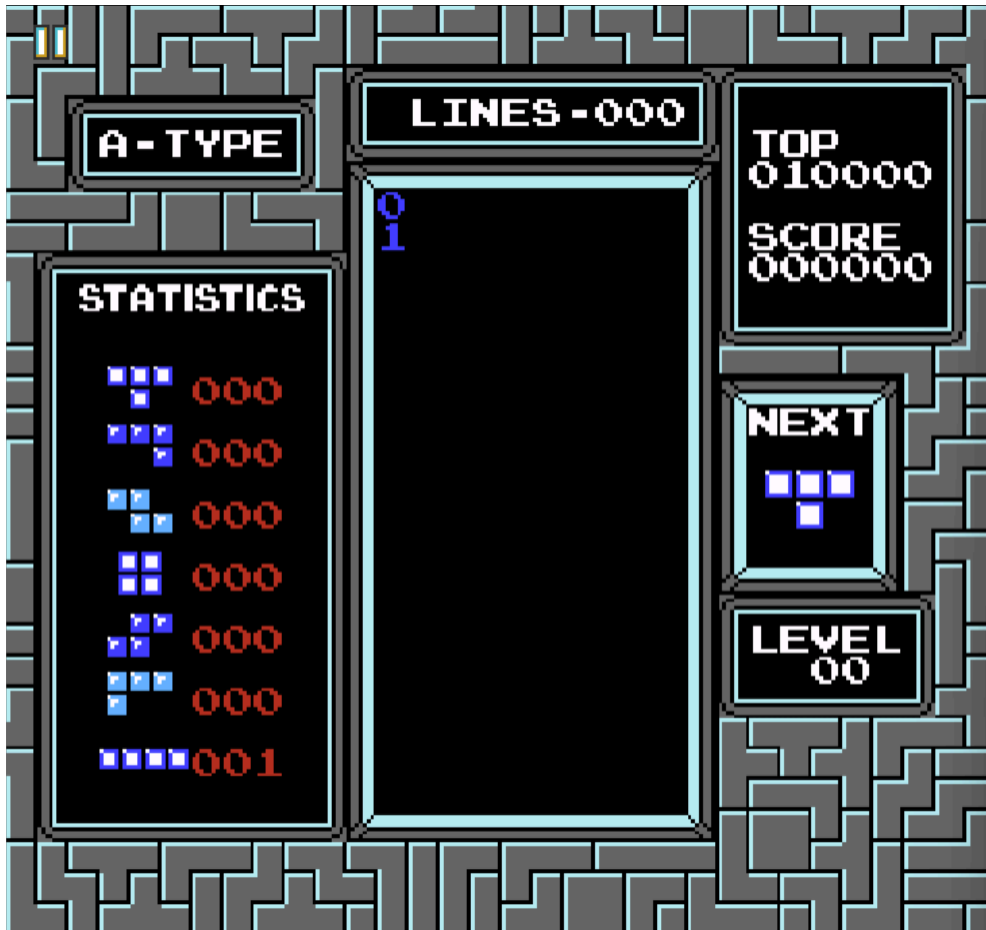
Disassembled *is_position_valid* function

```
2 void is_position_valid?(void)
3
4 {
5     byte bVar1;
6
7     tetrimino_iterator = (char)current_piece * '\x04';
8     bVar1 = (char)current_piece * '\f';
9     center_block_x_position = '\x04';
10    do {
11        if (0x15 < (byte)((&orientation_table)[bVar1] + tetriminoX._1_1_ + '\x02' +
12                        CARRY1((&orientation_table)[bVar1],tetriminoX._1_1_))) {
13            orientation_table_index = 0xff;
14            return;
15        }
16        center_block_y_position = (&orientation_table)[bVar1] * '\x02';
17        levelOrHeight =
18            (&orientation_table)[bVar1] * '\n' +
19            tetriminoX._1_1_ * '\n' + (char)tetriminoX +
20            CARRY1(tetriminoX._1_1_ * '\b',tetriminoX._1_1_ * '\x02');
21        if (*(byte *) (playFieldAddr +
22                      (ushort)(byte)((&orientation_table)[(byte)(bVar1 + 2)] + levelOrHeight))) < 0xef)
23        {
24            orientation_table_index = 0xff;
25            return;
26        }
27        if (9 < (byte)((&orientation_table)[(byte)(bVar1 + 2)] + (char)tetriminoX)) {
28            orientation_table_index = 0xff;
29            return;
30        }
31        bVar1 = bVar1 + 3;
32        center_block_x_position = center_block_x_position + -1;
33    } while (center_block_x_position != '\0');
34    orientation_table_index = 0;
35    return;
```

What it essentially does is go through the graphics table (aka the *_orientation_table* discussed earlier) and go over each block in the tetromino, ensuring that their X and Y values are within certain bounds. The index into the graphics table is stored in a single byte, and is incremented by 3 bytes so that it points to the next block in the loop. However, this is where all the trouble begins.

This worked fine when *_orientation_table* only had 19 rotation values, which at 12 bytes per rotation state adds up to 228 bytes worth of entries. However, now that our game had 6 new states, our table now contained 300 bytes worth of entries. This led to index overflows once we looked at piece IDs > 0x14, causing our O and I pieces not being displayed properly, which caused certain parts of the *is_position_valid* function to think we had gone out of bounds, or topped-out, and the game would end the level.

Modified game attempting to spawn in an I-Tetromino



While a very small error, solutions to this were anything but simple and ultimately, in the end, not capable of addressing all our issues.

Attempted Solutions

1) New Graphics Table

One approach had us rethinking how the `orientation_table` was structured. If we could make the table itself smaller, then we wouldn't have to have the index reach so high and we could avoid overflows. We recognized how the 3-bytes-per-block structure could be reduced to 2-bytes-per-block if we removed duplicate sprite-ID bytes and only kept the X and Y positions. A theoretical approach is shown below.

Theoretical Graphics Table

```
_orientation_table:
.byte $00, $ff, $00, $00, $00, $01, $ff, $00, $ff, $00, $00, $00, $00, $01, $01, $00 ; T-BLOCK
.byte $00, $ff, $00, $00, $00, $01, $01, $00, $ff, $00, $00, $00, $00, $ff, $01, $00
.byte $00, $ff, $00, $00, $00, $01, $ff, $ff, $ff, $00, $00, $00, $01, $00, $ff, $01 ; J-BLOCK
.byte $00, $ff, $00, $00, $00, $01, $01, $01, $ff, $00, $00, $00, $01, $00, $01, $ff
.byte $ff, $ff, $00, $00, $00, $01, $ff, $00, $ff, $01, $00, $00, $00, $01, $01, $00 ; Z-BLOCK
.byte $00, $ff, $00, $00, $01, $00, $01, $01, $01, $ff, $00, $00, $00, $ff, $ff, $00
.byte $00, $ff, $00, $00, $ff, $00, $ff, $01, $ff, $00, $00, $00, $00, $01, $01, $01 ; S-BLOCK
.byte $01, $ff, $00, $00, $01, $00, $00, $01, $ff, $ff, $00, $00, $00, $ff, $01, $00
.byte $ff, $01, $00, $00, $00, $ff, $00, $01, $ff, $00, $00, $00, $01, $00, $01, $01 ; L-BLOCK
.byte $00, $ff, $00, $00, $00, $01, $01, $ff, $ff, $ff, $00, $00, $ff, $00, $01, $00
.byte $ff, $fe, $ff, $ff, $ff, $00, $ff, $01, $fe, $00, $ff, $00, $00, $01, $00 ; I-BLOCK
.byte $00, $fe, $00, $ff, $00, $00, $00, $01, $fe, $ff, $ff, $ff, $00, $ff, $01, $ff
.byte $00, $ff, $00, $00, $01, $ff, $01, $00 ; O-BLOCK

_tiles:
.byte $7b, $7d, $7c, $7d, $7c, $7b, $7b
```

Whether the tile IDs would've been included in the table or not was still being discussed. Either way, this would've worked brilliantly, reducing our bytes down to 207. However, this would've required not just replacing the table, but reworking all the existing logic around accesses to the table. This effort was only magnified by one of the first constraints, that we couldn't just add in new instructions wherever we wanted. Not only would we have to make new logic (which also was in conflict with one of our general project approaches), but we'd either have to carefully add in the new instructions or replace several functions entirely. With all this in mind, we realized this wasn't a viable solution with the time remaining.

2) Graphic Table Offsets

Our next idea was to have a variable that stored either the start of the *orientation_table* or a +0xFF offset into that table, which would allow us to keep most of the logic the same and would only involve minor changes to the *rotation_table* and *orientation_table*. However, this required in-depth knowledge of assembly language and we still would've likely had to replace entire functions. Overall, while this was a much simpler solution than the table, getting it to work effectively was far more difficult than anticipated, and thus we had to abandon this method as well. Some pseudocode of what the function would've done is provided below.

Theoretical Table Offset Correction Code



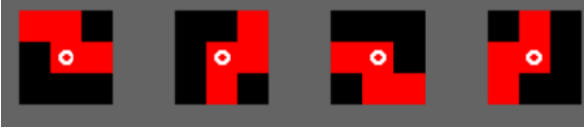




```
_update_table_index:
    lda #$69
    sta $0e
    lda #$fd
    sta $0f
    lda #$42 ; pieceorientation
    cmp #$14
    bcc @no_overflow
    lda #$fe
    sta $0f
@no_overflow:
    lda a:_nextIDtoSprite,X      ; $8BDC BD E5 8B
    rts
```

Spawning Solution

While the above methods had great merit to them, by the time we found the error and came up with these solutions, the difficulty in implementation paired with the limited amount of development time remaining meant we had to scrap both and come up with a more limited solution. While our final solution does solve almost everything, we unfortunately had to remove I-tetrominos from the game.

Recall that when we were first modifying the *rotation_table* and *orientation_table*, we realized that the placement of the O-piece in those tables was very frustrating, because while all the other pieces had 4 rotations each, the O-piece only had 1, adding an irritating shift to all the bytes after it and making it difficult to quickly tell exactly what indices were manipulating which pieces. To address this, we moved the O-piece to the end of the tables, making everything more aligned. However, the overflow bugs meant we couldn't spawn both I-tetrominoes and O-tetrominoes due to their high piece IDs. To solve this, we restructured the tables to have the O-piece have an ID of 0x14 (instead of 0x18), and to have the I-pieces start at 0x15. This gave us the following rotation values:

Final Rotation Values

Rotation Value	Associated Rotation
0x0, 0x1, 0x2, 0x3	
0x4, 0x5, 0x6, 0x7	
0x8, 0x9, 0xa, 0xB	
0xC, 0xD, 0xE, 0xF	
0x10, 0x11, 0x12, 0x13	
0x14	
0x15, 0x16, 0x17, 0x18	

Furthermore, we prevented the spawn of I-tetrominoes by manipulating our spawning tables to spawn O-tetrominoes instead of I-tetrominoes. This solution allowed us to spawn 6/7 pieces with our new rotation system, all without the game kicking us out the level, and all it took was slightly modifying our tables. The only downside was that now you couldn't play Tetris with I-pieces (which I suppose would demand renaming the game to “*Tetrs*”). While it's not the desired solution, given our limited assembly experience and time remaining, it was the best and only option.

Wall-Kicks

Wall-kicks are an addition to later SRS revisions that allow pieces to be “pushed” into place, even if the act of rotating a piece doesn’t perfectly fit. This provides the player with a little bit more flexibility with how they can actually fit pieces together, and for pieces to still rotate even if they’re pushed all the way against the wall.

Wall-Kicking example, see red Z-piece before (left) and after (right)



In this example, rotating a Z piece allows it to twist into place where originally it could not have possibly fit. The way these pieces check for available places to be pushed into is specified by the SRS guideline.

Since NES tetris was made before any sort of SRS guideline, it doesn’t have the wall-kicking specifications that SRS enforced. In fact, it has no wall-kicking capabilities whatsoever. If a tetromino is rotated and there are any overlapping blocks when it tests the rotation, it’ll cancel the action entirely.

NES Rotation

The NES game’s rotation code is stored at address 0x88AB. To summarize how it functions, the game determines which way you’re rotating the piece – to the left or right – and pulls the rotation information from the rotation table. It checks the position, and if that position does not break the game rules, the rotation persists. If the rotation breaks game rules, it reverts the rotation information back.

Rotation code in NES tetris as approximated C (left) and assembly (right)

```

1 void rotate_tetrimino(void)
2
3
4 {
5     char cVar1;
6
7     originalY = (byte)current_piece;
8     if ((newButtons & 0x80) == 0x80) {
9         current_piece._0_1_ = (&rotation_table)[(byte)((byte)current_pi
10         cVar1 = (byte)current_piece == 0;
11         is_position_valid?();
12         if (cVar1 != '\0') {
13             sfx = 5;
14             return;
15         }
16     }
17     else {
18         if ((newButtons & 0x40) != 0x40) {
19             originalY = (byte)current_piece;
20             return;
21         }
22         current_piece._0_1_ = (&rotation_table)[(byte)((byte)current_pi
23         cVar1 = (byte)current_piece == 0;
24         is_position_valid?();
25         if (cVar1 != '\0') {
26             sfx = 5;
27             return;
28         }
29     }
30     current_piece._0_1_ = originalY;
31     return;
32 }

```

Wall-Kick Implementation

Initially our idea for implementing kicks was to directly modify the rotation code itself. However, as mentioned earlier, unless byte alignment is perfect, the game refuses to run. Looking at the first two instructions of the rotation code, we can see that they take up 4 bytes total. By replacing the first instruction with a jump to a custom function at the end of ROMo (3 bytes) and the second instruction with a carry bit clear (1 byte), we can match the initial byte size and grant us flexibility with implementation length.

Our next roadblock came in the form of *shift_tetrimino*. The code that determines how pieces move laterally across the board. We thought we could utilize the function within our custom rotation code to simulate the checks that the tetrimino could be pushed towards. However, this wasn't a feasible solution as it contains too much button-parsing to be usable in the state it currently is in, so the piece placement testing had to be written from scratch. This made the implementation significantly more convoluted than initially planned.

SRS wall-kicking data is a fairly complex series of tests that depend on what piece is being rotated, what the initial state of the piece is, and what the requested state of the piece is going to be. All of these influence directionality of the tests required a personalized set of tests for just the I-piece on its own. Each of the coordinates in the

figure below describe what X and Y movements were made to the piece in each test. O signifies that the piece must be in a spawned state, R signifies the piece is rotated right, L signifies left, and 2 signifies 2 turns from spawn (moving downwards).

Tables denoting tests for ensuring valid wall-kicks

J, L, S, T, Z Tetromino Wall Kick Data

	Test 1	Test 2	Test 3	Test 4	Test 5
O->R	(0, 0)	(-1, 0)	(-1,+1)	(0,-2)	(-1,-2)
R->O	(0, 0)	(+1, 0)	(+1,-1)	(0,+2)	(+1,+2)
R->2	(0, 0)	(+1, 0)	(+1,-1)	(0,+2)	(+1,+2)
2->R	(0, 0)	(-1, 0)	(-1,+1)	(0,-2)	(-1,-2)
2->L	(0, 0)	(+1, 0)	(+1,+1)	(0,-2)	(+1,-2)
L->2	(0, 0)	(-1, 0)	(-1,-1)	(0,+2)	(-1,+2)
L->O	(0, 0)	(-1, 0)	(-1,-1)	(0,+2)	(-1,+2)
O->L	(0, 0)	(+1, 0)	(+1,+1)	(0,-2)	(+1,-2)

I Tetromino Wall Kick Data

	Test 1	Test 2	Test 3	Test 4	Test 5
O->R	(0, 0)	(-2, 0)	(+1, 0)	(-2,-1)	(+1,+2)
R->O	(0, 0)	(+2, 0)	(-1, 0)	(+2,+1)	(-1,-2)
R->2	(0, 0)	(-1, 0)	(+2, 0)	(-1,+2)	(+2,-1)
2->R	(0, 0)	(+1, 0)	(-2, 0)	(+1,-2)	(-2,+1)
2->L	(0, 0)	(+2, 0)	(-1, 0)	(+2,+1)	(-1,-2)
L->2	(0, 0)	(-2, 0)	(+1, 0)	(-2,-1)	(+1,+2)
L->O	(0, 0)	(+1, 0)	(-2, 0)	(+1,-2)	(-2,+1)
O->L	(0, 0)	(-1, 0)	(+2, 0)	(-1,+2)	(+2,-1)

Wall-Kick Implementation Problems

The kicks require both a lot of extra data and additional instructions, which eat up available storage on the ROM. As a result of that, implementations had to be small as well as functional, which further imposed additional challenges.

Bugs in the implementation while it was being developed were very hard to solve, as pieces would behave erratically or disappear altogether, and much of the debugging process was filtering out cycles dedicated to screen drawing or sound effects.

In the end, the wall-kicking code was too difficult for us to implement in time. We had too many roadblocks and ended up pivoting to exclusively finalizing the piece rotation table.

Final Product

In conclusion, we were able to understand a vast amount of the original Tetris assembly code! We dissected how rotation states are stored, displayed, and spawned, as well as how piece and boundary collisions work! We also figured out how to modify graphical data through software like YY-CHR (though we never had a need to). Overall, we're very proud of the work we've done and while we didn't hit all of our goals, we feel that we did a great job tackling this grandiose set of tasks.

Our modifications led to a more realized version of the 1989 classic, with a complete table of rotation states and the founding of new wall-kicking mechanics. While you can't spawn all 7 tetrominoes, the ones that you can play with are far more modernized!

For things we'd like to improve on next time, we would ideally minimize the *orientation_table* to prevent collision check overflows, finalize wall-kicking, fix the statistics window on the left, fix some minor graphical bugs, and add in graphics showing our names on the start-up screen as the ones responsible for modifications. That's it for now, thanks for reading!

P.S.

If you give us really good grades we'll totally add your name in the title screen! Totally not a bribe at all though