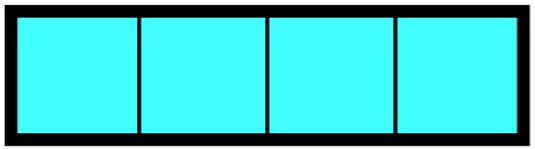
Tetroxide - Tetris in Rust

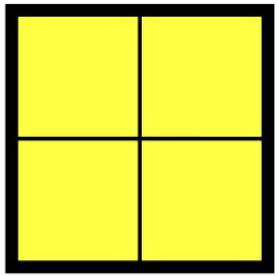
Eric Hamilton, Andrew Idak, Jesse Pingitore

Real Quick - Terminology

"**Tetris**" - Clearing four lines simultaneously with an "I" tetrimino.



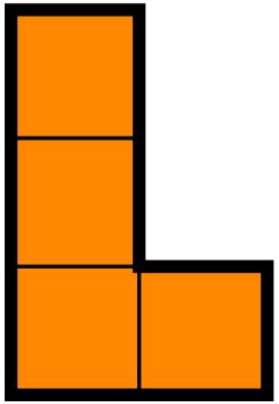
← "**I**" Block



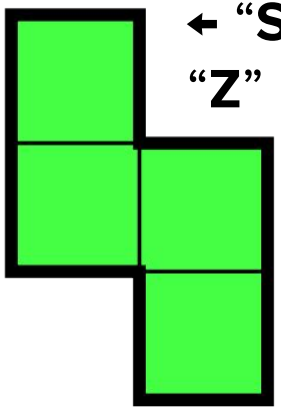
← "**Mino**"

← "**Tetrimino**", aka "Piece" or "Block"

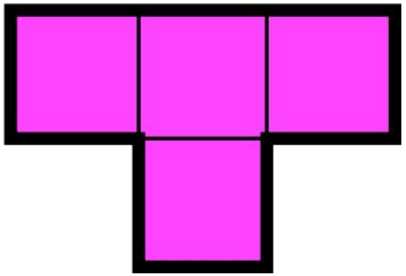
↓ "**L**" Block



← "**S**" or "**Z**" Block



↑ "**O**" Block



← "**T**" Block

[1] By Anypodetos - Own work based on: Tetromino Tiling 5x8.svg by R. A. Nonenmacher., CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=8663361

Tetroxide in A Nutshell

- Tetris, with total* memory safety!
  - *As far as TUI is memory safe.
- Follows Super Rotation System (**SRS**) rules
  - Governs how more intricate details of tetromino generation, orientation, rotation
    - I.E. pushing off of walls
    - locking in place
- Follows Guideline Scoring System
  - Standard Tetris scoring with additional bonuses for various T-spins and chaining combos.
- Standard “Endless Play” rules
  - Tetrimino velocity accelerates discreetly with time
  - Game continues until player makes enough errors to fill the 21st line with a mino.

Implementation Structure

- Crate::Tetris
  - Contains all essential game logic
    - SRS rule enforcement
    - Move scoring
    - Piece generation/dropping
  - [Some] unit testing as well
- Crate::Tetroxide
  - Contains all game management functionality
    - *View* and *Control* of the *Model/View/Control* architecture
    - Frame management
    - UI presentation & interaction handling via TUI

Why Rust?

- **Enums** succinctly match & describe rotations without playing with coordinates.
- Stacked **pattern matching** allows us to handle vast number of rotation cases far easier than with *if* condition checking.
- Code blocks anywhere allows us to eval and return **closure** computations directly from the matches.
- The small closures use functional programming patterns with ***into_iter*** and ***map*** to replace verbose loop rolling.
- ***Unreachable!*** macros allow us to succinctly handle impossible edge cases which the compiler isn't smart enough to understand.

```
// These are the different "origin" states we will be testing.
    let origin = if let Tetromino::I = self.tetromino {
        match (clockwise, new_rotation) {
            (true, State::Up) | (false, State::Right) => (row - 1, col),
            (true, State::Right) | (false, State::Down) => (row, col + 1),
            ...
        }
    } else {
        (row, col)
    };
    let mut origins = vec![origin];
    ...
    let kick_data_i1 = vec![(-2, 0), (1, 0), (-2, -1), (1, -2)];
    ...
    // We extend our possible tests with the 4 additional tests:
    origins.extend(
        match self.tetromino {
            Tetromino::O => return false, /* O Tetromino's have no rotational logic. */
            Tetromino::I => match (self.rotation, new_rotation) {
                // CW from Spawn State OR CCW to Inverted Spawn State
                (State::Up, State::Right) | (State::Left, State::Down) => kick_data_i1,
                // CCW to Spawn State OR CW from Inverted Spawn State
                (State::Right, State::Up) | (State::Down, State::Left) => {
                    kick_data_i1.into_iter().map(|(x, y)| (-x, -y)).collect()
                }
                ...
                _ => unreachable!(), /* THIS SHOULD NEVER HAPPEN. */
            },
            ...
        }
        .into_iter()
        .map(|(x, y)| (row + y, col + x)),
    );
}
```

Why (not) Rust? [Challenges]

- TUI was technically functional but logistically awful to work with
- Prototyping in Rust in was difficult due to library support and typing/memory safety constraints
    - Resulted in only fulfilling MVP, not reaching more stretch goals
    - Rust is strong with a sound idea, but not when sounding ideas

A sample of **TUI**'s pseudo-scripting ➜

```
let all = Layout::default()
    .direction(Direction::Horizontal)
    .constraints(
        [
            Constraint::Length((size.width - 48) / 2),
            Constraint::Length(48),
            Constraint::Length((size.width - 48) / 2),
        ]
        .as_ref(),
    )
    .split(size);
let layout = Layout::default()
    .direction(Direction::Horizontal)
    .constraints(
        [
            Constraint::Length(12),
            Constraint::Length(24),
            Constraint::Length(12),
            Constraint::Percentage(100),
        ]
        .as_ref(),
    )
    .margin(1)
    .split(all[1]);
let stats_layout = Layout::default()
    .direction(Direction::Vertical)
    .constraints([
        Constraint::Length(4),
        Constraint::Length(4),
        Constraint::Length(3),
        Constraint::Length(3),
        Constraint::Percentage(100),
    ])
    .split(layout[0]);
```

Demonstration

See <demo.gif> in the project directory!

... (google slides doesn't like 20MB .gif files)