

# Copilot Instructions for tactile-win

This document serves as comprehensive context for AI development assistants working on the `tactile-win` project, consolidating architecture and functional specifications.

---

## Project Overview

A Rust-based Windows native application inspired by GNOME’s Tactile extension for grid-based window management. The application provides a modal interface for quickly positioning and resizing windows using keyboard shortcuts and visual grid overlays.

## System Objective

A resident application that:

- Detects all connected monitors.
  - Defines an independent grid per monitor, initially with predefined dimensions and later configurable by the user.
  - Enters a modal mode through a global hotkey.
  - Displays an overlay over all screens with grids labeled by keys.
  - Allows selecting a rectangle using keyboard letters (QWERTY-like layout).
  - Repositions and resizes the active window according to the selection.
- 

## Recommended Tech Stack

To ensure optimal performance and maintainability, the following dependencies and technologies are strongly recommended:

### Core Dependencies

#### `windows-rs`

- **Primary Win32 Interface:** Use Microsoft’s official `windows-rs` crate for all Win32 API interactions
- **Version:** Latest stable release (0.52+)
- **Rationale:** Official Microsoft binding, comprehensive API coverage, actively maintained
- **Usage:** All platform layer modules must use this crate exclusively for Win32 calls

### Window Management

- **Custom HWND Creation:** Implement overlay windows using direct `windows-rs` Win32 calls
- **No External Window Libraries:** Avoid `winit` or similar as they add unnecessary complexity for overlay windows
- **Direct Control:** Full control over window styles, positioning, and behavior

### Rendering Backend

- **Primary Choice:** `tiny-skia` Pure Rust 2D graphics library
- **Performance:** Significantly faster than GDI for overlay rendering
- **Features:** Anti-aliasing, transparency, hardware acceleration
- **Alternative:** `skia-safe` if more advanced graphics features are needed
- **Avoid:** GDI/GDI+ (legacy, slow), Direct2D (complex setup)

### Platform-Specific Considerations

#### DPI Awareness

- **Critical Requirement:** Application must be DPI-aware to handle modern Windows displays
- **API Requirements:** Use `GetDpiForMonitor`, `GetSystemMetrics` with DPI context
- **Scaling Support:** Handle 100%, 125%, 150%, 200%+ scaling factors
- **Per-Monitor DPI:** Support different DPI values across multiple monitors
- **Initialization:** Include manifest file or call `SetProcessDpiAwarenessContext` in `main.rs` startup
- **Coordinate Virtualization:** Without proper DPI awareness, Windows will provide “fake” scaled coordinates

Window Styles for Overlay

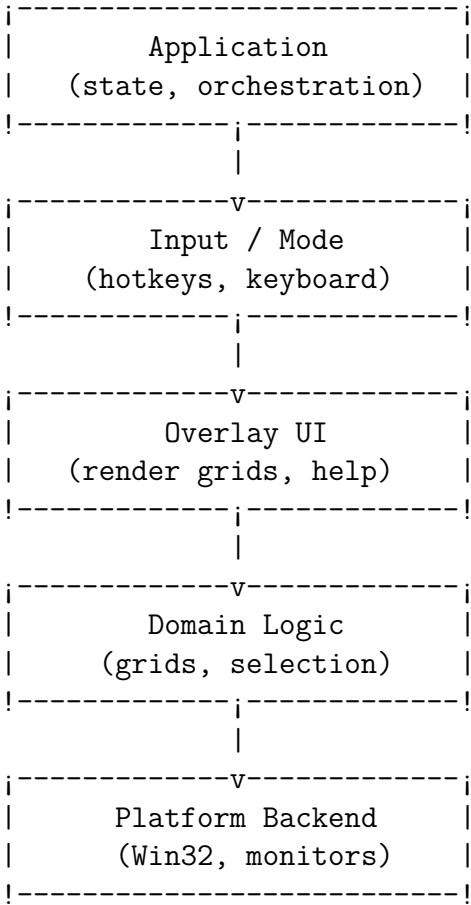
- **WS\_EX\_NOACTIVATE**: Overlay must not steal focus from active window
  - **WS\_EX\_TOPMOST**: Ensure overlay appears above all other windows
  - **WS\_EX\_LAYERED**: Enable transparency and alpha blending
  - **WS\_EX\_TOOLWINDOW**: Prevent overlay from appearing in taskbar
  - **Z-Order Management**: Other topmost applications (Task Manager, etc.) can hide overlay
  - **Keep-Alive Strategy**: Implement periodic z-order refresh during selection mode
- 

Architecture and Design Principles

Core Design Principles

1. Strict separation of responsibilities.
2. Domain logic independent from Win32 whenever possible.
3. Win32 access encapsulated in well-defined modules.
4. Explicit application states.
5. Avoid monolithic modules.
6. Modular and testable architecture.

Layered Architecture



Project Organization

```
src/
|-- main.rs
|-- app/
|   |-- mod.rs
|   |-- state.rs
|   |-- controller.rs
|-- domain/
|   |-- mod.rs
|   |-- grid.rs
|   |-- keyboard.rs
|   |-- selection.rs
|-- platform/
|   |-- mod.rs
|   |-- windows.rs
|   |-- monitors.rs
```

```
|  |-- window.rs
|-- ui/
|  |-- mod.rs
|  |-- overlay.rs
|  |-- render.rs
|-- input/
|  |-- mod.rs
|  |-- hotkeys.rs
|  |-- keyboard.rs
!-- config/
|  |-- mod.rs
|  |-- settings.rs
```

---

## Module Responsibilities

### main.rs

- Application entry point.
- Initializes the app and the message loop.
- Contains no business logic.

### app/ Orchestration and global state

- Manages application state: `enum AppState { Idle, Selecting(SelectionState) }`
- Coordinates input, domain, UI, and platform layers.
- Performs no geometric calculations or direct Win32 calls.

### domain/ Pure, testable logic

- **grid.rs**: NxM grid representation, validation, coordinate conversion
- **keyboard.rs**: QWERTY layout mapping, key → grid index
- **selection.rs**: Multi-cell selections, normalization, bounding box calculation
- **Critical**: This module does not know about Win32 types or structures.

### platform/ Operating system interface (Windows)

- **monitors.rs**: Monitor enumeration, resolution/work area retrieval, **DPI awareness handling**
- **window.rs**: Active window retrieval, movement, resizing (`SetWindowPos`)
- **windows.rs**: General Win32 helpers, type conversions
- **Critical**: This is the only module strongly coupled to Win32.

## DPI Awareness Requirements

- **monitors.rs** must handle high-DPI displays using `GetDpiForMonitor`
- All coordinate calculations must account for per-monitor DPI scaling
- Grid positioning must be DPI-aware to prevent blurry or misaligned overlays
- Support for mixed-DPI environments (monitors with different scaling factors)

### ui/ Overlay and rendering

- **overlay.rs**: Overlay window creation/destruction, visibility control, **focus management**
- **render.rs**: Grid rendering, letter rendering, help legends
- Contains no business logic.

## Critical Focus Management

- **Non-Activating Overlay**: The overlay window must capture keyboard input without becoming the active window
- **Active Window Preservation**: The window being resized must remain the “active window” throughout the selection process
- **Window Style Requirements**: Use `WS_EX_NOACTIVATE` and `WS_EX_TOPMOST` to achieve proper focus behavior
- **Keyboard Capture**: Implement low-level keyboard hooks or window message filtering to capture keys without focus theft

input/ User input

- **hotkeys.rs**: Global hotkey registration, mode switching
- **keyboard.rs**: Key capture, event translation, monitor navigation
- Does not decide final actions, only reports events.

config/ Configuration and persistence

- Grid configuration per monitor, validation, loading/saving
- 

Functional Behavior

Grid System

The application divides screen(s) into NxM grids with the following characteristics:

- **Default**: 3x2 grid per monitor
- **Labeling**: QWERTY layout (Q,W,E,R,... = top row; A,S,D,F,... = second row)
- **Constraints**: Minimum cell size 480x360 pixels (suggested for usability)
- **Independence**: Each monitor has its own grid
- **Size Adaptation**: Reduce grid size within reasonable thresholds for smaller monitors
- **Minimum Display**: Reject configuration for displays smaller than 600px height

User Interaction Flow

1. **Activation**: Global hotkey triggers modal mode
2. **Overlay**: Grid overlay appears on all screens with letter labels
3. **Selection Start**: User presses first key (e.g., Q) -> this cell is highlighted as selection start
4. **Selection End**: User presses second key (e.g., S) -> this defines the selection rectangle
5. **Multi-cell**: Selection spans from start to end cell (e.g., Q -> S = cells Q,W,A,S)
6. **Multi-monitor**: Arrow keys navigate between screens before or during selection
  - **Navigation**: Left/Right arrow keys move between horizontally arranged monitors
  - **Visual Feedback**: Active monitor shows letter labels, inactive monitors show grid without letters
  - **Mid-Selection**: Monitor switching allowed during selection process
7. **Application**: Active window resized/repositioned to selected area
8. **Exit**: Return to idle mode

Selection Lifecycle Details

- **Two-Step Process**: Selection requires exactly two key presses (start -> end)
- **Visual Feedback**: First key press highlights start cell, second key shows final selection rectangle
- **Cancel Options**: ESC key or global hotkey again cancels selection
- **Single Cell**: Pressing same key twice selects single cell (e.g., Q -> Q = cell Q only)
- **Rectangle Formation**: Selection always forms rectangle from top-left to bottom-right of pressed keys
- **Order Independence**: Same result regardless of which corner is pressed first (Q -> S = S -> Q)
- **Invalid Input**: Invalid key combinations close overlay without effect
- **Timeout**: Overlay closes automatically after 30 seconds of inactivity
- **Window Compatibility**: Non-resizable windows (dialogs) cause overlay to close without effect

Example Interaction

4x2 grid: Q W E R  
          A S D F

User types: Q, S  
Result: Window occupies cells Q, W, A, S (top-left quadrant)

---

Technical Implementation

Execution Flow

User presses hotkey

```
--> input::hotkeys
--> app::controller
--> AppState::Selecting
--> ui::overlay::show
--> input::keyboard::capture
--> domain::selection
--> domain::grid
--> platform::monitors
--> platform::window::apply_rect
--> ui::overlay::hide
--> AppState::Idle
```

Key State Management

```
enum AppState {
    Idle,
    Selecting(SelectionState),
}
```

Critical Functional Rules

- Each monitor has independent grid
- Default grid: 3x2
- Minimum cell size: 480x360 pixels
- Two-step selection process (start key -> end key)
- Multi-cell selection creates bounding boxes
- Cross-monitor selection deferred to Phase 4 (advanced feature)
- QWERTY keyboard layout determines grid mapping
- Overlay must preserve active window focus
- All coordinates must be DPI-aware
- No interaction with Windows Snap Layouts (independent operation)
- Single desktop operation (no virtual desktop integration)
- Letter labels must be prominent, clear, and high-contrast
- No overlay animations (standard window resize animations only)

---

Development Guidelines

Implementation Order (Recommended)

Phase 1 – Infrastructure

1. Monitor enumeration
2. Window management
3. Logical grid implementation
4. Per-monitor minimum size validation

Phase 2 – Domain Logic

5. Keyboard layout
6. Cell selection (single monitor)
7. Bounding box calculation

Phase 3 – Interaction

8. Global hotkey
9. Basic overlay
10. Letter capture
11. Single-monitor window positioning

Phase 4 – Advanced Features

12. Cross-monitor selection and window movement
13. Persistent configuration
14. On-screen help
15. System tray integration
16. Final polish

## Cross-Monitor Complexity Warning

**Cross-monitor window movement** involves significant technical complexity and should be implemented only after the single-monitor MVP is stable:

- **Virtual Desktop Coordinates:** Windows uses virtual coordinate system where secondary monitors can have negative coordinates
- **Monitor Arrangement:** Left/right/above/below monitor positioning affects coordinate calculations
- **DPI Differences:** Each monitor may have different DPI scaling factors
- **Work Area Variations:** Taskbar and system UI positioning varies per monitor
- **Focus Complications:** Moving windows between monitors can trigger unexpected focus changes

**Recommendation:** Implement single-monitor functionality first, then add cross-monitor as an advanced feature in Phase 4.

## Architecture Constraints

- **Domain Purity:** Keep domain/ modules completely Win32-agnostic
- **Platform Encapsulation:** All Win32 calls must be within `platform/` modules
- **State Centralization:** Route all state changes through `app/controller.rs`
- **Separation of Concerns:** Each module has single, well-defined responsibility

## Testing Guidelines

All code must be properly tested following these layer-specific patterns:

### Domain Layer Testing

- **All critical domain logic must be covered by unit tests** as domain logic is pure and easily testable
- Test all grid calculations, keyboard mappings, and selection logic
- Property-based testing may be introduced later for geometric calculations
- Example test structure:

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn grid_cell_coordinates_are_calculated_correctly() {
        let grid = Grid::new(3, 2, 1920, 1080);
        assert_eq!(grid.cell_rect(0, 0), Rect { x: 0, y: 0, w: 640, h: 540 });
    }
}
```

### Platform Layer Testing

- **Mock Win32 APIs** where possible using traits and dependency injection
- Create integration tests for actual Win32 behavior in `tests/` directory
- Use conditional compilation for Windows-specific tests: `#[cfg(target_os = "windows")]`
- Platform layer tests focus on error handling and API contracts rather than exhaustive behavior coverage.

### Application Layer Testing

- Test state transitions exhaustively: `Idle <-> Selecting`
- Mock dependencies on domain and platform layers
- Verify coordination between layers without testing implementation details

### UI Layer Testing

- Focus on rendering logic, not actual graphics output
- Test overlay positioning calculations
- Mock rendering backends for unit tests

## Integration Testing

- Full end-to-end scenarios in `tests/integration/`
- Test complete user workflows: hotkey -> selection -> window positioning
- Use virtual monitors and mock windows for CI environments

## Functional Requirements

The application must:

- Detect number of connected screens and their resolutions
- Initialize each screen with default 3x2 grid
- Prevent configuration of grids with cells smaller than 480x360 pixels
- Support multi-cell and cross-monitor selections
- Provide overlay legend for help and configuration access

---

This design prioritizes clarity, maintainability, and extensibility. Any new feature should be integrated while respecting the responsibilities and constraints defined in this document.