Why Refactor: A Post-Mortem Analysis

Original Application Goals

The eframe-paint application aimed to create a modern, web-capable painting application using Rust and egui. The core features included:

- Layer-based drawing and image manipulation
- Multiple tool support (brush, eraser, selection)
- Transform operations with visual gizmos
- Undo/redo functionality
- Image import capabilities

Core Problems Encountered

1. State Management Complexity

The application ran into significant state management issues, particularly evident in the gizmo implementation. The root causes were:

- **Scattered State**: Tool state, document state, and UI state were intermingled without clear boundaries
- **Implicit State Transitions**: No formal state machine meant tools could interfere with each other
- **Mutable State Access**: Multiple components needed mutable access to shared state, leading to complex borrow patterns
- **State Persistence**: Difficulty in determining what state should persist between sessions

2. Tool Lifecycle Management

The tool system revealed several architectural weaknesses:

- **Unclear Tool Boundaries**: Tools could affect document state directly without proper encapsulation
- **State Interference**: Tools could leave residual state that affected other tools
- **Missing Cleanup**: No formal activation/deactivation lifecycle for tools
- **Incomplete State Tracking**: Tool state wasn't fully captured in the undo/redo system

3. Transform Gizmo Challenges

The gizmo implementation became our canary in the coal mine, exposing deeper architectural issues:

- **State Ownership**: Unclear ownership of transform state between the gizmo and layer
- **Event Handling**: Complex interaction between mouse events, tool state, and document state
- **Visual Update Timing**: Difficulty coordinating visual updates with state changes
- **Undo/Redo Complexity**: Challenge of capturing complete transform state for history

Immediate Mode GUI Impact

Advantages in Our Context

1. **Simple State Synchronization**: Immediate mode naturally handles keeping UI in sync with application state
2. **Easy Prototyping**: Quick iteration on UI layouts and interactions
3. **Automatic Layout**: Simplified UI positioning and scaling
4. **Cross-Platform**: Works well across desktop and web targets

Challenges and Limitations

1. **State Management Overhead**:
    - Need to maintain application state separately from UI
    - Must carefully manage state updates to prevent flickering
    - Complex to maintain state between frames
2. **Performance Considerations**:
    - Continuous redrawing can be CPU intensive
    - Need careful management of texture resources
    - Challenge in handling large numbers of draw calls
3. **Tool Implementation Complexity**:
    - Difficult to maintain tool state between frames
    - Complex to implement dragging operations
    - Challenge in implementing precise pixel-level operations
4. **Architectural Tensions**:
    - Immediate mode philosophy conflicts with traditional document editing patterns
    - Difficulty in implementing retained-mode concepts like selection handles
    - Challenge in managing long-running operations

Retained Mode Alternative Analysis

A retained mode approach could offer several advantages:

Potential Benefits

1. **Natural Document Model**:
    - Better alignment with document editing metaphor
    - Clearer separation of document and UI state
    - More straightforward persistence model
2. **Simplified State Management**:
    - Clearer ownership of state
    - More natural undo/redo implementation
    - Better tool state isolation
3. **Performance Advantages**:

    - More efficient rendering of static content
    - Better caching opportunities
    - Reduced CPU usage for unchanged areas

Potential Drawbacks

1. **Implementation Complexity**:
    - Need to build more UI infrastructure
    - More complex event handling system
    - Higher initial development cost
2. **Platform Limitations**:
    - More challenging to target web platform
    - Potential performance issues on low-end devices
    - More complex cross-platform story

Architectural Lessons for Paint Applications

1. State Management

- Implement a formal state machine for application modes
- Clearly separate document, tool, and UI state
- Use event system for state changes
- Consider command pattern for all state modifications

2. Tool System

- Define clear tool lifecycle (activate, use, deactivate)
- Isolate tool state from document state
- Implement proper cleanup on tool switches
- Use command pattern for tool operations

3. Document Model

- Separate document model from view/controller logic
- Implement clear layer management system
- Use proper serialization strategy
- Consider using Entity Component System (ECS) for document elements

4. Transform Operations

- Implement transform operations as self-contained systems
- Use proper math models for transforms
- Consider matrix stack approach
- Implement proper pivot point handling

Path Forward

Option 1: Refactor Current Codebase

Pros:

- Preserve existing functionality
- Incremental improvement
- Lower initial effort

Cons:

- Constrained by existing architecture
- May not solve fundamental issues
- Technical debt may persist

Option 2: Fresh Start

Pros:

- Clean architectural foundation
- Better separation of concerns
- Modern best practices from start

Cons:

- Higher initial effort
- Need to reimplement existing features
- Potential project delays

Recommendation

Based on the analysis, a fresh start would be more beneficial in the long term. The current codebase has served as an excellent prototype, surfacing important architectural considerations and requirements. A new implementation could:

1. Start with a proper state machine
2. Implement a clean tool system
3. Use an event-driven architecture
4. Properly separate concerns
5. Consider alternative UI approaches

The lessons learned from the current implementation are invaluable for informing the design of a more robust solution.