CONDUCT: Coordinated Orchestration of Navigation Dynamics for Unified Control and Tracking

Technical Documentation Team
YipYap Project
favicon.pdf

September 6, 2025

Abstract

This paper details the design and implementation of YipYapś CONDUCT system, a comprehensive architecture developed to ensure responsive and fluid navigation within the applicationś image gallery. We describe the interplay of key composables: `ScrollManager` for direct DOM interaction, `useScrollCoordinator` for centralized request orchestration and conflict resolution, and `useScrollPerformanceMonitor` for real-time performance analytics. The system prioritizes user experience through passive event listeners, debounced scroll handling, and an intelligent conflict resolution mechanism for various scroll sources. By integrating these components, the CONDUCT system achieves high performance, prevents UI jank, and provides robust diagnostics for complex scroll interactions in a SolidJS environment.

1 Introduction

Modern web applications, especially those handling rich media like image galleries, demand highly responsive and jank-free scrolling experiences. In the YipYap project, the image gallery serves as a central component, necessitating a sophisticated scroll management system that can gracefully handle diverse user inputs, programmatic scrolls, and ensure seamless visual feedback. This paper outlines the architectural decisions and technical implementation of the YipYap CONDUCT system, focusing on its modular design, performance optimizations, and comprehensive monitoring capabilities.

Traditional scrolling implementations often suffer from race conditions, UI freezes, and inconsistent behavior when multiple sources attempt to control the scroll position simultaneously. The YipYap CONDUCT system addresses these challenges through a layered approach, centralizing scroll requests and providing robust mechanisms for conflict resolution and performance observation.

2 CONDUCT System Architecture

The YipYap CONDUCT system is built upon a foundation of interconnected SolidJS composables, each responsible for a specific aspect of scroll management. This modularity enhances maintainability, testability, and allows for fine-grained control over the scrolling behavior.

2.1 ScrollManager: The Direct DOM Interface

At the lowest level of the CONDUCT architecture is the `ScrollManager` class (defined in `src/components/Gallery/ScrollManager.tsx`). This class is responsible for direct interaction with the DOMś scrollable element. Its primary functions include:

- Maintaining the current scroll position (`currentScrollY`).
- Providing scroll bounds (minimum and maximum scroll positions) via `getScrollBounds()`.
- Implementing a `smoothScrollTo` method for animating scroll transitions, incorporating an ease-in-out quad easing function for natural motion and utilizing `requestAnimationFrame` for smooth rendering. The `scrollTimeout` parameter, provided during initialization, determines the duration of this animation.
- Handling cleanup of animation frames to prevent memory leaks and ensure smooth transitions with `cleanup()`.
- Exposing `isActive` to indicate if a smooth scroll animation is in progress.

`ScrollManager` acts as the singular source of truth for the raw scroll position and the direct manipulator of the scrollable DOM element.

2.2 useScrollCoordinator: The Orchestration Layer

The `useScrollCoordinator` composable (defined in `src/composables/useScrollCoordinator.ts`) serves as the central orchestration layer for all scroll requests within the application. It receives scroll requests from various sources (e.g., keyboard navigation, mouse wheel, auto-scrolling) and manages their execution based on defined priorities.

Key responsibilities of the `useScrollCoordinator` include:

- **Request Queuing:** Maintains a queue of pending scroll operations (`queuedOperations`), ensuring that requests are processed in an orderly fashion, with a configurable `maxQueueSize` (default: 10).

- **Priority-Based Conflict Resolution:** Assigns numerical priorities to different scroll types (defined in `SCROLL_PRIORITIES`) to prevent race conditions and ensure the most relevant scroll operation is executed. Higher-priority requests can cancel lower-priority ones.
- **Debouncing and Throttling:** The `debounceMs` parameter (default: 16ms) primarily controls the processing frequency of the scroll operation queue, rather than directly debouncing raw scroll events. This ensures that scroll operations are processed at a consistent rate, preventing excessive updates. Additionally, a `userScrollTimeout` (managed by `userScrollTimeoutId`) is used to accurately detect the cessation of user-initiated scrolling, distinguishing it from programmatic scrolls. The `debounceTimeoutId` manages general debouncing for internal operations to ensure smooth UI responsiveness.
- **User Scroll Detection:** Differentiates between user-initiated scrolls (`userScrolling` signal) and programmatic scrolls to enable intelligent conflict resolution and maintain user control. It uses a `scrollTimeout` to determine the end of a user scroll.
- **Integration with ScrollManager:** Utilizes the `ScrollManager` to perform the actual DOM scroll operations via an `internalScrollCallback` passed to `requestScroll`.
- **Integration with Performance Monitor:** Feeds detailed scroll operation data to the `useScrollPerformanceMonitor` via methods like `startScrollOperation`, `endScrollOperation`, and `cancelScrollOperation` for real-time analysis.
- **Initialization:** The `init(element)` method is crucial for attaching scroll listeners to the gallery element. This method, along with internal helper functions like `initializeGalleryElement` and `setupScrollListeners`, ensures that the `useScrollCoordinator` is properly connected to the DOM and responsive to scroll events.
- **Request Parameters:** Each scroll request can include `force` (to bypass active animation checks), `onCompletion` (a callback executed upon the operationś success or failure), and `timeout` (a configurable duration after which the operation is considered failed if not completed).

All scroll requests from application components are routed through this coordinator, providing a single control point for scroll behavior.

2.3 useScrollPerformanceMonitor: Tracking and Diagnostics

The `useScrollPerformanceMonitor` composable (defined in `src/composables/useScrollPerformanceMonitor.ts`) is dedicated to real-time performance analytics and debugging of scroll interactions. It integrates with the `ScrollManager` to observe scroll events and track various metrics, and leverages `usePerformanceMonitor` for broader performance insights.

Its features include:

- **Operation Tracking:** Records the `id`, `type` (e.g., keyboard, wheel, auto-scroll, manual), `source`, `startTime`, `endTime`, `duration`, `actualY`, `cancelled`, and `conflictedWith` of each scroll operation.
- **Performance Metrics:** Calculates and reports key indicators such as `totalScrollOperations`, `averageScrollDuration`, `longestScrollDuration`, `scrollsPerSecond`, `totalConflicts`, `conflictRate`, `criticalConflicts`, `interruptedUserScrolls`, `scrollJankEvents`, and `frameDropsDuringScroll`.
- **Conflict Detection:** Employs `detectConflicts()` to identify instances where multiple scroll operations overlap in time, target, or type, indicating potential race conditions. Conflicts are categorized by `severity` (low, medium, high, critical).
- **Performance Metrics (texttttScrollPerformanceMetrics):** Calculates and reports key performance indicators such as `totalScrollOperations`, `averageScrollDuration`, `longestScrollDuration`, `scrollsPerSecond`, `totalConflicts`, `conflictRate`, `criticalConflicts`, `interruptedUserScrolls`, `scrollJankEvents`, and `frameDropsDuringScroll`.
- **Warning System (texttttScrollWarning):** Generates warnings for various issues like slow scrolls, jank events, and memory spikes, with different `severity` levels.
- **Memory Usage Tracking:** Monitors JavaScript heap memory usage during active scrolling periods using a `createEffect` and `performanceMonitor.measureMemoryUsage()` (an asynchronous call), updating the `memoryUsage` signal periodically.
- **Thresholds (texttttTHRESHOLDS):** Defines configurable thresholds for scroll duration, conflict rates, and frame drops to categorize and warn about performance issues. These include `SCROLL_DURATION_WARNING` (100ms) and `CRITICAL` (300ms), `CONFLICT_RATE_WARNING` (10%) and `CRITICAL` (30%), and `FRAME_DROP_WARNING` (3 dropped frames) and `CRITICAL` (10 dropped frames), along with `OPERATIONS_PER_SECOND_WARNING` (10 ops/sec) and `CRITICAL` (20 ops/sec).
- **Debugging Utilities:** Provides `exportData()` for detailed JSON output of metrics, operations, conflicts, and warnings, and `getScrollStatus()` for a high-level overview of monitoring activity.

This proactive monitoring approach is critical for maintaining a high-quality user experience as the application evolves and scales.

2.4 useGalleryScroll: The Gallery Interface

The `useGalleryScroll` composable (defined in `src/composables/useGalleryScroll.tsx`) acts as the primary interface for gallery-specific scrolling functionalities. It ties together the `ScrollManager`, `useScrollCoordinator`, and `useScrollPerformanceMonitor` to provide a cohesive scrolling experience within the image gallery, integrating with the `useGallery` context.

Key features provided by `useGalleryScroll` include:

- **Scroll to Selected Item (
texttt scrollToSelected):** Automatically scrolls the gallery to ensure the currently selected image is within the viewport. It calculates the `targetY` considering a 15% margin buffer zone at the top and bottom of the viewport. It also includes a sophisticated post-scroll position verification mechanism: after an automatic scroll, a `setTimeout` is initiated, followed by a `debouncedRAF` re-request if misalignment is detected, to correct any lingering discrepancies and ensure the selected item is perfectly in view. This uses the `autoScrolling` signal for status.

- **Smooth Scroll Animations (
texttt smoothScroll):** Exposes a method that wraps `scrollCoordinator.requestScroll` to animate scrolls to specific target positions.

- **Wheel Event Handling (
texttt setupWheelHandler):** Manages mouse wheel events for navigation. It prevents default browser scrolling (using { `passive: false` } to allow `preventDefault`) and enables custom image navigation by changing the selected image index based on `e.deltaY`. It then requests a `wheel` type scroll through the `scrollCoordinator`, triggering a `smoothScrollTo` on the `ScrollManager` and a `gallery.select` call. The system currently uses `e.deltaY` directly for both mouse wheel and touchpad input, without specific separate handling for `touchpadDelta`.

- **Position Checking (
texttt startPositionChecking):** Initiates a `window.setInterval` (default: 500ms) to periodically verify the position of selected items within the viewport. It requests `position-check` scroll operations via the coordinator, which may lead to corrective `scrollToSelected` calls if misalignment is detected.

- **RAF-Throttled UI Updates (
texttt debouncedRAF):** This helper function acts as a throttling mechanism, utilizing `requestAnimationFrame` to ensure that computationally intensive UI updates and animation frame requests are batched and executed optimally. This prevents excessive calls and ensures smooth rendering by synchronizing updates with the browserś rendering cycle, rather than simply debouncing them.

This composable abstracts away the complexities of the underlying scroll system, providing a clean API for gallery interactions and ensuring selected items are always in view.

2.4.1 ImageViewer Core: Zoom and Pan

Beyond the main gallery scroll, the `ImageView` component (`src/components/ImageViewer/ImageView.tsx`) provides granular control over image display through sophisticated zooming and panning capabilities. This is especially crucial for high-resolution images where users need to inspect details. Key mechanisms include:

- **Smooth Zooming:** Implemented via the `onWheel` event, it adjusts the `scale` of the image. A reduced `deltaY` factor (`-0.001`) ensures gradual transitions, and cursor-relative calculations maintain the focal point during zooming. Scale limits (1x to 5x) are enforced.
- **Advanced Panning:** Achieved by tracking `onMouseDown`, `onMouseMove`, and `onMouseUp` events to update the `position` of the image via CSS `transform`. When zoomed in, users can drag the image to pan, with the cursor changing to `grab`/`grabbing` states for visual feedback.
- **Minimap Navigation:** A minimap, whose viewport is updated via `updateMinimapViewport` and `createEffect` hooks, provides real-time tracking of the visible area. Users can interact with the minimap via `handleMinimapClick` to quickly navigate large images.
- **Double-Click Reset:** A `onDblClick` handler (`handleDoubleClick`) allows users to quickly reset the zoom and position to their default (1x zoom, centered).

This core image viewer interacts with the broader scroll system indirectly. While `ImageView` handles the internal navigation of a single image, the `useImageViewerState` composable (`src/composables/useImageViewerStat` manages the modes (e.g., `labelling`, `captionFocused`) that can influence or restrict zooming and panning. For instance, during a labeling session, direct image manipulation might be temporarily paused or altered to avoid conflicts, ensuring a coordinated user experience.

2.4.2 Image Loading and Progressive Processing

Efficient image loading is paramount for a smooth gallery scrolling experience, preventing jank and ensuring images are displayed as quickly as possible. YipYap employs a progressive loading strategy, significantly enhanced by the `useProgressiveLoading` composable (`src/composables/useProgressiveLoading.ts`). This composable is not directly responsible for image fetching but provides a robust mechanism for processing large datasets and performing time-sliced operations without blocking the UI thread.

- **Time-Sliced Processing:** `useProgressiveLoading` enables the application to break down large tasks (such as image data processing or `localStorage` operations) into smaller, manageable chunks. This allows the browser to yield control back to the UI thread between processing batches, maintaining responsiveness.
- **Jank Prevention:** By preventing long-running JavaScript tasks, this composable directly contributes to a jank-free scrolling experience, as image data can be processed in the background without causing UI freezes.

- **Optimized LocalStorage Operations:** It also provides utilities for saving and loading data from `localStorage` progressively, ensuring that even large cached datasets do not impede UI performance during critical operations like scrolling.

This progressive processing, in conjunction with the image prioritization (as briefly mentioned in `ImageView.tsx` for `thumbnail_img` and `preview_img`), ensures that image assets are loaded and prepared efficiently, contributing to the overall fluidity of the galleryś scrolling and navigation.

2.4.3 GalleryContext: Global State and Scroll Influence

The `GalleryContext` (`src/contexts/gallery.ts` and `src/contexts/GalleryContext.tsx`) serves as the central hub for the applicationś global gallery-related state. This context is crucial for scrolling behavior, as changes in its state directly influence what is displayed and how the `useGalleryScroll` composable reacts.

- **Selected Item Tracking:** The `selected` signal within `GalleryContext` tracks the currently active image in the gallery. Any change to this `selected` index immediately triggers the `scrollToSelected` function in `useGalleryScroll`, ensuring the newly selected image is brought into view.
- **Data Management:** `GalleryContext` manages the overall `items` data, which dictates the size and content of the scrollable area. As items are loaded, filtered, or reordered, the scrollable bounds and targets for automatic scrolling are dynamically updated.
- **Path and Pagination:** The `path` and `page` states within the context define the current directory and pagination, respectively. Navigating through different paths or pages can necessitate significant scroll adjustments, coordinated by the `useScrollCoordinator`.

Thus, `GalleryContext` acts as the data backbone, providing the necessary information and reactive triggers for the entire CONDUCT scrolling system to maintain a coherent and responsive user experience.

2.4.4 Keyboard Navigation and Selection Integration

User experience is significantly enhanced by robust keyboard navigation, especially in a gallery setting where quick traversal through items is essential. The `useSelection` composable (`src/contexts/selection.ts`) plays a pivotal role in enabling keyboard-driven scrolling and item selection.

- **Grid-Based Navigation:** `useSelection` manages the logic for navigating through gallery items using arrow keys (up, down, left, right). It intelligently calculates the next selected item based on a virtual grid layout, even if items are not perfectly aligned visually.

- **Automatic Scrolling on Selection:** When a new item is selected via keyboard, `useSelection` triggers the `scrollToSelected` function from `useGalleryScroll`. This ensures that the newly focused item is always brought into the viewport, providing a seamless browsing experience without manual scrolling.
- **Multi-Selection Shortcuts:** It supports advanced keyboard shortcuts like `Ctrl+A` for selecting all items and `Shift+Arrow` for extending a multi-selection. While these dont́ directly trigger scrolls, they interact with the selection state, which in turn can influence the scroll position if the selected area moves outside the viewport.
- **Pagination and Virtualization Awareness:** `useSelection` is designed to work with paginated data and the virtual selection system. It ensures that keyboard navigation correctly accounts for loaded and visible items, and that selections persist across different pages or viewports.

This tight integration of keyboard controls with the core scrolling and selection mechanisms ensures that users can efficiently and comfortably navigate large galleries without relying solely on mouse or touch input.

2.4.5 Responsive Design and Dynamic Layout Adjustments

In a dynamic application like YipYap, layouts frequently adjust due to various factors, including window resizing, panel toggling, or changes in content. Ensuring that the scrolling system adapts seamlessly to these changes is critical for a consistent user experience. The `ResizeObserver` API is strategically utilized across the application to facilitate responsive design and prevent layout-induced scrolling issues.

- **Dynamic Column Measurement:** The `measure_columns` directive (`src/directives.tsx`) employs `ResizeObserver` to detect changes in the number of CSS grid columns within the gallery. This allows the application to dynamically adjust its grid-based navigation logic (managed by `useSelection`) and ensure that keyboard navigation accurately reflects the current layout, implicitly affecting scroll targets.
- **Image Viewer Adaptation:** Within the `ImageView` component (`src/components/ImageViewer/ImageView.tsx`), `ResizeObserver` is used to update the minimapś viewport and image display parameters when the container size changes. This ensures that zooming and panning calculations remain accurate and the displayed image scales correctly, preventing visual anomalies that could lead to incorrect scroll perceptions.
- **Bounding Box Editor Responsiveness:** The `BoundingBoxEditor` component (`src/components/ImageViewer/B` also leverages `ResizeObserver` to dynamically adjust the canvas size for drawing bounding

boxes. This ensures that the editing area remains synchronized with the image viewerś dimensions, preventing misalignment and ensuring a smooth editing experience, which can indirectly impact perceived scroll consistency during interactive operations.

By actively monitoring element dimensions and reacting to changes, `ResizeObserver` helps maintain the integrity of the UI layout, which in turn supports the accuracy and fluidity of the scrolling system across various screen sizes and dynamic content configurations.

2.4.6 Error Handling and Notification System Integration

The YipYap application incorporates a comprehensive error handling and notification system to provide users with timely feedback on critical events, including those related to scrolling performance. While the `useScrollPerformanceMonitor` primarily focuses on internal diagnostics and logging, its generated `ScrollWarning`s are consumed by other parts of the application, particularly for debugging and, implicitly, for user awareness through a broader notification mechanism.

- **ScrollWarning Generation:** The `useScrollPerformanceMonitor` actively detects and generates `ScrollWarning` objects for issues such as slow scrolls (`SCROLL_DURATION_WARNING/CRITICAL`), high conflict rates (`CONFLICT_RATE_WARNING/CRITICAL`), and frame drops (`FRAME_DROP_WARNING/CRITICAL`). These warnings include details about their type, severity, and a descriptive message.
- **Debugging and Monitoring Overlay:** The `DebugOverlay` component (`src/components/Debug/DebugOverlay.t` directly subscribes to the `scrollCoordinator.getScrollWarnings()` to display real-time performance warnings and metrics. This provides developers with immediate visual feedback during development and testing.
- **Implicit User Notification:** Although `useScrollPerformanceMonitor` does not directly call `appContext.notify` (the primary user-facing notification system), the presence of these warnings in the debug overlay and their severity levels align with the applicationś general philosophy of providing clear feedback. Critical scroll performance issues, if severe enough to impact user experience, would implicitly be surfaced through other UI elements or, if a direct integration were implemented, via `appContext.notify` to inform the user.

This layered approach ensures that while granular performance data is available for debugging, the groundwork is laid for potential user-facing notifications should scroll-related issues reach a critical threshold, maintaining the applicationś commitment to a high-quality user experience.

3 Key Mechanisms and Performance Deep Dive

3.1 Priority-Based Conflict Resolution

One of the most critical aspects of the CONDUCT system is its ability to manage and resolve conflicts between concurrent scroll requests. The `useScrollCoordinator` assigns numerical priorities to different scroll operation types:

- **Manual (1000):** Highest priority, typically from direct user interaction (e.g., dragging the scrollbar or direct touch input).
- **Keyboard (800):** High priority, for keyboard-driven navigation (e.g., arrow keys).
- **Wheel (700):** High priority, for mouse wheel events for navigation.
- **Auto-Scroll (500):** Medium priority, for programmatic scrolls initiated to keep selected items in view.
- **Smooth-Scroll (400):** Medium priority, for general smooth animation requests initiated by the application.
- **Position-Check (200):** Lowest priority, for background checks and minor corrections to item visibility.

When a new scroll request is made, the coordinator compares its priority with any currently active operation. Higher-priority requests can cancel lower-priority ones, ensuring that user intent is always respected and preventing janky transitions. This mechanism is crucial for a smooth and predictable user experience, especially when automated scrolls might conflict with manual user input.

3.1.1 Conflict Resolution Strategies

Beyond simple priority comparison, the `useScrollPerformanceMonitor` (specifically the `determineResolutionStrategy` function) employs specific rules to resolve conflicts, ensuring a logical and user-centric behavior:

- **Manual Scroll Precedence:** If a new operation is a `manual` scroll, it will always attempt to cancel any conflicting `auto-scroll` operations, prioritizing direct user input.
- **Preserving Manual Scrolls:** If an existing active operation is a `manual` scroll, any new conflicting operation (except other manual scrolls of even higher priority, which is rare) will be cancelled to preserve the userś ongoing interaction.
- **Keyboard Over Auto-Scroll:** Keyboard navigation (`keyboard` type) takes precedence over `auto-scroll` operations. A new keyboard scroll will cancel conflicting auto-scrolls, and an active keyboard scroll will cause new auto-scrolls to be cancelled.

- **Position Check De-prioritization:** `position-check` operations are generally cancelled if they conflict with any other scroll type, as they are background tasks with the lowest priority.
- **Default Resolution:** For other conflicts where no specific rule applies, the general strategy is to cancel older conflicting operations in favor of the newest request.

This detailed approach to conflict resolution ensures that the system remains responsive and predictable even under complex, concurrent scroll scenarios.

3.2 Passive Event Listeners for Jank Prevention

To optimize performance, particularly for high-frequency events like `scroll` and `wheel`, the CONDUCT system strategically leverages passive event listeners. By adding { `passive: true` } to event listeners (e.g., on the gallery element for general scrolling), the browser is informed that the event handler will not call `preventDefault()`. This allows the browser to perform its default scrolling behavior immediately, without waiting for the JavaScript handler to complete, thereby significantly reducing jank and improving responsiveness, especially on mobile devices.

However, itś important to note the exception in `setupWheelHandler`. Here, { `passive: false` } is explicitly used because the handler *does* call `e.preventDefault()` to prevent the default page scrolling behavior, allowing for custom image navigation via the wheel. This demonstrates a careful balance between performance optimization and desired user interaction.

3.3 Real-time Performance Monitoring and Debugging

The deep integration of `useScrollPerformanceMonitor` provides unprecedented insights into the scroll systemś behavior. The `PerformanceDashboard` component, in conjunction with this monitor, can display real-time metrics, active operations, and a detailed log of conflicts and warnings. This powerful combination allows developers to:

- **Identify Bottlenecks:** Pinpoint specific scroll operations that are causing performance issues or exceeding defined thresholds.
- **Debug Race Conditions:** Understand precisely when and why scroll conflicts occur, providing the necessary data to refine priority rules or execution logic.
- **Optimize Transitions:** Analyze `averageScrollDuration`, `longestScrollDuration`, `scrollJankEvents`, and `frameDropsDuringScroll` to refine animation timings, easing functions, and overall scroll fluidity.
- **Monitor Memory Usage:** Detect potential memory leaks or excessive memory consumption related to active scrolling periods, which is vital for long-running applications.

- **Proactive Issue Detection:** The warning system (`ScrollWarning`) provides early alerts for potential performance degradations before they impact the user significantly.

This proactive monitoring approach is critical for maintaining a high-quality user experience as the application evolves and scales.

3.4 Leveraging SolidJS Reactivity

SolidJŚ fine-grained reactivity plays a crucial role in the high-performance nature of CONDUCT. Unlike frameworks that rely on virtual DOM diffing, SolidJS directly updates the DOM based on reactive state changes. This means:

- **Minimal Re-renders:** Only the specific parts of the DOM affected by a scroll-related state change are re-rendered, reducing unnecessary computation.
- **Efficient DOM Updates:** Direct manipulation of DOM properties (like `scrollTop` by `ScrollManager`) combined with SolidJŚ reactivity ensures updates are applied efficiently.
- **Optimized Effects:** `createEffect` in `useScrollPerformanceMonitor` for memory tracking, and `createMemo` for `getScrollMetrics`, ensure that computations are only performed when their dependencies change, further optimizing performance.

This inherent efficiency of SolidJS complements the CONDUCT architecture, contributing to a jank-free user experience.

3.4.1 Virtual Selection for Performance Optimization

For galleries containing a large number of items, directly applying styles to every selected item can lead to significant performance degradation. The `useVirtualSelection` composable (`src/composables/useVirtualSelectio` addresses this by implementing a virtual selection mechanism. This system optimizes rendering performance by:

- **Intersection Observer-Based Visibility:** It uses the `Intersection Observer API` to detect which gallery items are currently visible within the viewport. Selection styles are only applied to these visible items.
- **Deferred Style Application:** Style updates for selected items are deferred and applied in batches, preventing the blocking of the main thread and ensuring smooth scrolling, especially when many items become visible or invisible simultaneously.
- **Configurable Thresholds:** Virtual selection is enabled only when the total number of items exceeds a configurable `enableThreshold` (default: 50), balancing performance optimization with simpler rendering for smaller datasets.

- **Automatic Cleanup:** Resources associated with off-screen items are automatically cleaned up, further reducing memory footprint and improving responsiveness.

By dynamically applying selection styles only to elements that are actively in view, `useVirtualSelection` significantly reduces the rendering load, contributing to a smoother and more performant scrolling experience, particularly in large galleries.

3.5 Animation and Timing Precision

Smooth animations are paramount for a good scrolling experience. CONDUCT achieves this through:

- **`requestAnimationFrame`:** Utilized by `ScrollManager.smoothScrollTo` and `debouncedRAF` to ensure that animations are synchronized with the browserś rendering cycle, preventing visual tearing and maximizing smoothness.
- **Cubic-Bezier Easing:** The `smoothScrollTo` method employs a `easeInOutQuad` easing function, providing a natural acceleration and deceleration profile to scroll animations, enhancing perceived smoothness.
- **Debounced UI Updates:** The `debouncedRAF` helper ensures that computationally intensive UI updates (like position corrections) are batched and executed optimally, preventing them from blocking the main thread.

These precise timing and animation controls contribute significantly to the perceived responsiveness and fluidity of the galleryś scrolling.

3.6 Comparison with Other Scroll Management Systems

YipYap's CONDUCT system exhibits a sophisticated, multi-layered approach to scroll management, distinguishing itself from many common implementations by emphasizing coordinated control, priority-based conflict resolution, and extensive performance monitoring.

3.6.1 Similarities and Differences with General Web Scrolling Practices

- **Asynchronous vs. Synchronous Scrolling:** Many traditional web applications, especially those relying heavily on `scroll` event listeners, can suffer from "Blank Spots" or jank due to the asynchronous nature of browser scrolling [1]. CONDUCT, like advanced systems, aims for a more controlled, synchronous-like experience by managing scroll operations centrally. While CONDUCT uses `passive: false` for the wheel handler to allow `preventDefault()`, indicating a degree of synchronous control over that specific input, it also leverages `requestAnimationFrame` for smooth animations, aligning with modern best practices for jank prevention [3, 4].

- **Debouncing and Throttling:** The use of `debounceMs` in `useScrollCoordinator` for processing the scroll queue and `userScrollTimeout` for detecting user-initiated scrolling aligns with common patterns for optimizing scroll performance [2, 4]. This prevents excessive event firing and ensures that heavy computations don't block the main thread.

- **`requestAnimationFrame` for Animations:** CONDUCT's reliance on `requestAnimationFrame` for its `smoothScrollTo` method is a standard and highly recommended practice for achieving smooth animations synchronized with the browser's rendering cycle [4].

- **Passive Event Listeners:** CONDUCT's strategic use of passive event listeners for general scrolling events (`passive: true`) where `preventDefault()` is not called, and `passive: false` for specific cases like wheel events where `preventDefault()` is necessary, demonstrates a nuanced understanding of performance optimization [1, 5]. This is a critical aspect highlighted in modern web development for avoiding scroll-blocking.

- **Virtualization:** While the search results highlight libraries like `hyperlist` [7] and `fastgrid` [6] that offer virtual scrolling for rendering millions of rows, YipYap's `useVirtualSelection` composable provides a similar benefit by only applying selection styles to visible items based on `Intersection Observer API` to optimize rendering performance for large galleries. This is a common pattern in high-performance UIs.

3.6.2 Unique Strengths of YipYap's CONDUCT

- **Multi-layered Architecture:** CONDUCT's clear separation of concerns into `ScrollManager` (DOM interface), `useScrollCoordinator` (orchestration), `useScrollPerformanceMonitor` (diagnostics), and `useGalleryScroll` (gallery-specific interface) provides a highly modular and maintainable system. This level of architectural detail and dedicated components for each aspect of scrolling is more comprehensive than many general-purpose scroll utilities.

- **Priority-Based Conflict Resolution:** The detailed priority system (Manual ¿ Keyboard ¿ Wheel ¿ Auto-Scroll ¿ Smooth-Scroll ¿ Position-Check) and specific conflict resolution strategies in `useScrollCoordinator` and `useScrollPerformanceMonitor` are a significant differentiator. This sophisticated handling of concurrent scroll requests ensures user intent is prioritized and prevents erratic behavior often seen in simpler implementations.

- **Comprehensive Performance Monitoring:** The `useScrollPerformanceMonitor` with its detailed operation tracking, metrics calculation (total operations, average duration, conflicts, jank events, frame drops, memory usage), and warning system (`ScrollWarning`) provides exceptionally deep insights for debugging and optimizing scroll performance. This level of built-in diagnostics is not typically found in generic scroll libraries.

- **SolidJS Fine-Grained Reactivity:** CONDUCT leverages SolidJS's unique fine-grained reactivity, which directly updates the DOM without a virtual DOM, leading to minimal re-renders and efficient DOM updates. This framework-specific optimization contributes significantly to CONDUCT's high-performance nature, distinguishing it from React-based solutions that still contend with virtual DOM diffing [1]. Libraries like Hydroxide [9] and Starbeam [8] also focus on fine-grained reactivity for performance, aligning with SolidJS's approach.

3.6.3 Comparison with Specific Libraries/Tools

- **Solid Primitives `@solid-primitives/scroll` [10]:** This library provides basic reactive primitives for scroll position tracking (`createScrollPosition`, `useWindowScrollPosition`). CONDUCT builds upon these fundamental concepts by adding orchestration, conflict resolution, and performance monitoring layers, making it a much more complete and application-specific scroll management system.
- **`Lenis` [11]:** Lenis is a smooth scroll library that aims to create immersive interfaces and normalize user inputs. It addresses issues with scroll-linked animations caused by multi-threading. While `Lenis` focuses on the *smoothness* and *feel* of scrolling, CONDUCT provides a more comprehensive *management* system that includes prioritization, conflict resolution, and detailed performance diagnostics, in addition to smooth animations.
- **`HyperList` [7] and `fast-grid` [6]:** These are virtual scrolling libraries designed for rendering millions of rows efficiently. While CONDUCT incorporates virtual selection (`useVirtualSelection`), its scope is broader, encompassing various scroll inputs, conflict resolution, and performance monitoring, rather than solely focusing on large list rendering. `fast-grid` also mentions a custom event loop to prioritize tasks and never drop a frame, which resonates with CONDUCT's `useScrollCoordinator`'s request queuing and priority management.
- **`scrollen` [12] (React Hook):** This is a performant utility scroll hook for React that provides scroll position, direction, and programmatic scrolling functions. It's similar in concept to some of CONDUCT's individual composables, but CONDUCT's integrated system with conflict resolution, performance monitoring, and SolidJS-specific optimizations goes beyond a simple utility hook.
- **`solid-custom-scrollbars` [13] and `solid-virtual-scroll` [14]:** These are SolidJS-specific libraries for custom scrollbars and virtual scrolling, respectively. They address specific aspects of scroll UI and performance within the SolidJS ecosystem. CONDUCT, however, is a higher-level architectural system that *might* leverage such libraries for specific UI/performance needs, but its core functionality is about managing the overall scroll behavior of the application.

4 Algorithms

4.1 Priority-Based Scroll Operation Management

The core algorithm for managing scroll operations is based on a priority queue system:

```
Algorithm: Priority-Based Scroll Operation Management
Input: ScrollRequest (type, source, targetY, callback)
Output: Operation success/failure

1. Generate unique operation ID
2. Determine priority level based on type:
   - Manual scroll: 1000
   - Keyboard navigation: 800
   - Wheel navigation: 700
   - Auto-scroll: 500
   - Smooth scroll: 400
   - Position check: 200

3. Check for conflicts:
   For each active operation A:
     If abs(A.startTime - newOp.startTime) < 50ms:
       If abs(A.targetY - newOp.targetY) > 100px OR
          (A.type is manual AND newOp.type is auto) OR
          (A.type is auto AND newOp.type is manual):
         Record conflict
         If A.priority > newOp.priority:
           Cancel newOp
           Return failure
         Else:
           Cancel A
           Continue

4. If queue.length >= maxQueueSize:
   Remove lowest priority operation

5. Add operation to queue
6. If no current operation:
   Execute next operation
7. Return operation ID
```

4.2 Smooth Scroll Animation

```
Algorithm: Smooth Scroll Animation
Input: targetY, currentY, duration
Output: Continuous scroll position updates

1. Initialize:
   startTime = performance.now()
   startY = currentY
   distance = targetY - startY
   currentVelocity = 0
   dampingFactor = 0.85

2. Animation loop:
   While not completed:
     currentTime = performance.now()
     elapsed = currentTime - startTime
     progress = Math.min(elapsed / duration, 1)

     // Cubic bezier easing
     t = progress
     easedProgress = t * t * (3 - 2 * t)

     // Calculate new position with momentum
     targetPosition = startY + (distance * easedProgress)
     currentPosition = currentY

     // Apply velocity with damping
     velocity = (targetPosition - currentPosition) * dampingFactor
     currentVelocity = currentVelocity * 0.9 + velocity * 0.1

     // Update position
     newY = currentPosition + currentVelocity

     // Apply bounds
     newY = Math.max(0, Math.min(newY, maxScroll))

     // Update scroll position
     element.scrollTop = newY

     // Check completion
     if progress >= 1 and abs(newY - targetY) <= tolerance:
       break

```

```
    requestAnimationFrame(loop)
```

4.3 Scroll Conflict Detection

The algorithm for detecting and resolving scroll conflicts:

```
Algorithm: Scroll Conflict Detection
Input: newOperation, activeOperations
Output: Conflict information or null

1. Initialize:
   conflictingOps = []
   timeThreshold = 50ms
   distanceThreshold = 100px

2. For each activeOp in activeOperations:
   If activeOp.id !== newOperation.id:
     // Check timing overlap
     timeOverlap = abs(activeOp.startTime - newOperation.startTime) < timeThreshold

     // Check target position conflict
     targetConflict = false
     If activeOp.targetY exists and newOperation.targetY exists:
       targetConflict = abs(activeOp.targetY - newOperation.targetY) > 100px

     // Check type conflicts
     typeConflict = (
       (activeOp.type === "manual" && newOperation.type === "auto-scroll") ||
       (activeOp.type === "auto-scroll" && activeOp.type === "manual") ||
       (activeOp.type === "position-check" && newOperation.type !== "position-check")
     )

     If timeOverlap && (targetConflict || typeConflict):
       Add activeOp to conflictingOps

3. If conflictingOps is not empty:
   Create conflict = {
     id: generate\_unique\_id(),
     timestamp: performance.now(),
     operations: [newOperation, ...conflictingOps],
     severity: calculate\_severity(conflictingOps),
     description: generate\_description(newOperation, conflictingOps),
     resolutionStrategy: determine\_resolution\_strategy(newOperation, conflictingOps)
   }
   Return conflict

4. Return null
```

4.4 Selected Item Position Tracking

Algorithm for maintaining selected item visibility:

```
Algorithm: Selected Item Position Tracking
Input: selectedElement, galleryElement
Output: Scroll adjustment if needed

1. Initialize:
   galleryRect = galleryElement.getBoundingClientRect()
   selectedRect = selectedElement.getBoundingClientRect()

   // Calculate visible area with margins
   marginRatio = 0.15
   visibleTop = galleryRect.top + (galleryRect.height * marginRatio)
   visibleBottom = galleryRect.bottom - (galleryRect.height * marginRatio)

2. Check visibility:
   elementMostlyVisible = (
     selectedRect.top >= visibleTop - selectedRect.height * 0.5 &&
     selectedRect.bottom <= visibleBottom + selectedRect.height * 0.5
   )

3. If not elementMostlyVisible:
   needsScroll = (
     selectedRect.top < visibleTop ||
     selectedRect.bottom > visibleBottom ||
     selectedRect.top > visibleBottom ||
     selectedRect.bottom < visibleTop
   )

4. If needsScroll:
   targetY = Math.max(0,
     galleryElement.scrollTop +
     (selectedRect.top - galleryRect.top) -
     (galleryRect.height / 2) +
     (selectedRect.height / 2)
   )

   Return { needsScroll: true, targetY }
Else:
   Return { needsScroll: false }
```

4.5 Scroll Completion Detection

Algorithm for determining when a scroll operation has completed:

```
Algorithm: Scroll Completion Detection
Input: scrollOperation, element
Output: Completion status

1. Initialize:
   checkCount = 0
   maxChecks = 100  // Max 1 second of checking
   lastScrollTop = element.scrollTop
   lastCheckTime = performance.now()
   tolerance = 10  // 10px tolerance
   minSpeed = 0.1  // pixels per millisecond

2. While checkCount < maxChecks:
   currentY = element.scrollTop
   currentTime = performance.now()

   // Calculate scroll speed
   scrollDelta = abs(currentY - lastScrollTop)
   timeDelta = currentTime - lastCheckTime
   scrollSpeed = timeDelta > 0 ? scrollDelta / timeDelta : 0

   // Check completion conditions
   hasReachedTarget = abs(currentY - targetY) <= tolerance
   hasStoppedMoving = scrollSpeed < minSpeed

   If hasReachedTarget && hasStoppedMoving:
     Return { completed: true, success: true }

   // Update tracking variables
   lastScrollTop = currentY
   lastCheckTime = currentTime
   checkCount++

   Await nextFrame()
```

```

3. Return { completed: true, success: false, reason: "timeout" }
```

4.6 Wheel Event Processing

Algorithm for processing wheel events and determining scroll behavior:

```
Algorithm: Wheel Event Processing
Input: WheelEvent, currentSelectedIndex
Output: New selected index and scroll behavior

1. Initialize:
   preventDefault()
   items = gallery.data()?.items
   If !items: Return

2. Calculate new index:
   newSelectedIdx = currentSelected
   If event.deltaY > 0:  // Scroll down
     newSelectedIdx = min(currentSelected + 1, items.length - 1)
   Else:  // Scroll up
     newSelectedIdx = max(currentSelected - 1, 0)

3. Request scroll operation:
   scrollCoordinator.requestScroll({
     type: "wheel",
     source: deltaY > 0 ? "mouse-wheel-next" : "mouse-wheel-prev",
     callback: () => {
       // Calculate target scroll position
       element = querySelector(‘#gallery .item:nth-child(${newSelectedIdx + 1})‘)
       If !element: Return

       galleryRect = galleryElement.getBoundingClientRect()
       elementRect = element.getBoundingClientRect()

       targetY = max(0,
         galleryElement.scrollTop +
         (elementRect.top - galleryRect.top) -
         (galleryRect.height / 2) +
         (elementRect.height / 2)
       )

       // Perform scroll
```

```
        scrollManager.smoothScrollTo(targetY, true)
        gallery.select(newSelectedIdx)
      }
    })
```

These algorithms work together to create a cohesive scrolling system that handles various user interactions while maintaining performance and providing a smooth user experience. Each algorithm is designed to handle specific aspects of the scrolling system, from high-level coordination to low-level animation details.

5 Conclusion

The YipYap CONDUCT system represents a robust, highly performant, and observable solution for managing complex scroll interactions within a demanding image gallery application. By meticulously separating concerns among `ScrollManager`, `useScrollCoordinator`, `useScrollPerformanceMonitor`, and `useGalleryScroll`, the architecture achieves unparalleled modularity, maintainability, and debuggability.

The implementation of priority-based conflict resolution, strategic use of passive event listeners, and comprehensive real-time performance monitoring collectively ensure a smooth, responsive, and truly jank-free scrolling experience. The CONDUCT system sets a high standard for UI performance in SolidJS applications, demonstrating how a well-designed architectural approach can tackle complex interaction challenges while delivering an exceptional user experience.

6 References

References

[1] Doron, O. (2020). *Our journey to understand scrolling across different browsers.* monday Engineering. `https://engineering.monday.com/our-journey-to-understand-scrolling-across-different-browsers/`

[2] Quante, T. (2024). *Master Efficient Window Scroll Event Handling in JavaScript: Best Practices and Tips.* Tobi's Blog - qbit.me. `https://blog.q-bit.me/master-efficient-window-scroll-event-handling-in-javascript-best-practices-and-tips/`

[3] Hirota, Y. (2023). *A case study on scroll-driven animations performance.* Chrome for Developers. `https://developer.chrome.com/blog/scroll-animation-performance-case-study/`

[4] Jiang, S. (2015). *How to develop high performance onScroll event?*. joji.me. `https://joji.me/en-us/blog/how-to-develop-high-performance-onscroll-event/`

[5] Lawson, N. (2017). *Scrolling on the web: A primer*. Windows Blogs - Microsoft Edge Blog. `https://blogs.windows.com/msedgedev/2017/03/08/scrolling-on-the-web/`

[6] Petersson, G. (n.d.). *fast-grid: World's most performant DOM-based web table*. GitHub. `https://github.com/gabrielpetersson/fast-grid`

[7] Branyen, T. (n.d.). *hyperlist: A performant virtual scrolling list utility capable of rendering millions of rows*. GitHub. `https://github.com/tbranyen/hyperlist`

[8] Katz, Y. (n.d.). *Starbeam — Simple and Fun Reactivity*. StarbeamJS. `https://www.starbeamjs.com/`

[9] Hydroxide-js. (n.d.). *hydroxide: Next Generation Reactive JavaScript Framework*. GitHub. `https://github.com/hydroxide-js/hydroxide`

[10] Solid Primitives. (n.d.). *Scroll*. Solid Primitives Community. `https://primitives.solidjs.community/package/scroll/`

[11] Studio Freight. (n.d.). *Lenis – Get smooth or die trying*. `https://lenis.studiofreight.com/`

[12] Moureira, J. (n.d.). *scrollen: A performant utility scroll hook for React*. GitHub. `https://github.com/joaom00/scrollen`

[13] Diragb. (n.d.). *solid-custom-scrollbars: Custom Scrollbars for Solid*. GitHub. `https://github.com/diragb/solid-custom-scrollbars`

[14] SupertigerDev. (n.d.). *solid-virtual-scroll*. GitHub. `https://github.com/SupertigerDev/solid-virtual-scroll`