# Optimizing Zustand State Updates for Draggable Components in React

When implementing draggable UI elements with Zustand (vanilla, without middleware), careful state management is crucial for smooth interactivity. You want to avoid excessive re-renders or losing reactivity while updating an element’s **x/y** position in the store during drag operations. Below are best practices and techniques – including throttling, debouncing, and using refs – to ensure optimal performance and maintain a responsive drag experience.

## Use Atomic Selectors to Minimize Re-renders

Zustand allows components to subscribe to specific pieces of state (via selectors) instead of the entire store. **Always use selectors to pick just the needed state** (e.g. the x and y coordinates) rather than subscribing to the whole store. Subscribing to the entire store (or using a selector that returns a new object each time) will cause your component to re-render on any state change, even if unrelated[[1]](https://tkdodo.eu/blog/working-with-zustand#:~:text=2%20const%20,useBearStore)[[2]](https://tkdodo.eu/blog/working-with-zustand#:~:text=Effectively%2C%20this%20means%20that%20selectors,the%20content%20is%20the%20same). For example, prefer:

const x = useStore(state => state.x);  
const y = useStore(state => state.y);  
// or:  
const [x, y] = useStore(state => [state.x, state.y], shallow);

This ensures the component only updates when the relevant slice (x or y) changes. Zustand uses **useSyncExternalStoreWithSelector** under the hood, so it will compare the selected value on each update and only trigger a re-render if it has actually changed[[3]](https://philipp-raab.medium.com/zustand-state-management-a-performance-booster-with-some-pitfalls-071c4cbee17a#:~:text=2,directly%20affected%20by%20that%20state)[[4]](https://philipp-raab.medium.com/zustand-state-management-a-performance-booster-with-some-pitfalls-071c4cbee17a#:~:text=to%20a%20set%20of%20listeners,and%20subsequently%20calls%20all%20listeners). By keeping selectors *atomic* (returning primitive values or using shallow comparison for tuples/objects), you prevent unnecessary render cycles. In contrast, a selector that returns a new object each time (e.g. {x: state.x, y: state.y}) would always be seen as changed and force a re-render[[2]](https://tkdodo.eu/blog/working-with-zustand#:~:text=Effectively%2C%20this%20means%20that%20selectors,the%20content%20is%20the%20same).

Additionally, isolate high-frequency state from other global state. For instance, if your store holds many values, consider splitting the draggable positions into their own slice or even a separate store. This way, updating a drag position won’t invalidate selectors for unrelated data. In scenarios with **multiple draggable items**, structure your state to target updates granularly. For example, store positions in an object or map keyed by item ID. Then update only the changed item’s entry immutably, so that other items’ references remain unchanged. This prevents other components from re-rendering when one item moves[[5]](https://www.synergycodes.com/blog/guide-to-optimize-react-flow-project-performance#:~:text=The%20main%20issue%20arises%20from,are%20being%20dragged%20or%20not). Using Zustand’s selector, a component can subscribe to state.positions[itemId] – if the object reference for that item’s coordinates stays the same when others move, those other components won’t update.

## Throttle Frequent Drag Updates for Performance

Dragging triggers *many* pointermove events – often far more frequently than the browser can repaint frames[[6]](https://nolanlawson.com/2019/08/11/high-performance-input-handling-on-the-web/#:~:text=There%20is%20a%20class%20of,the%20browser%20can%20paint%20frames). Updating Zustand state on every single tiny movement can lead to janky performance or overwhelming React with updates. A common technique is **throttling** these updates so they happen at most, say, 60 times per second (roughly one per frame) or whatever frequency keeps the UI smooth.

**Why throttle?** If the user moves the mouse quickly, pointermove could fire multiple times between animation frames[[7]](https://nolanlawson.com/2019/08/11/high-performance-input-handling-on-the-web/#:~:text=Image%3A%20Screenshot%20of%20Chrome%20Dev,many%20as%20four%20pointermove%20events)[[8]](https://nolanlawson.com/2019/08/11/high-performance-input-handling-on-the-web/#:~:text=Again%2C%20let%E2%80%99s%20take%20a%20step,one%20we%20need%20to%20paint). We usually only care about the latest position by the time the next frame is drawn. Throttling ensures we drop intermediate updates and only apply the latest, preventing redundant work. For example, you can use requestAnimationFrame or a utility like Lodash’s throttle to limit how often you call setState for the coordinates. A pattern using requestAnimationFrame is especially effective: queue only the last movement per frame[[8]](https://nolanlawson.com/2019/08/11/high-performance-input-handling-on-the-web/#:~:text=Again%2C%20let%E2%80%99s%20take%20a%20step,one%20we%20need%20to%20paint). In practice:

let frame = 0;  
const onDragMove = (e) => {  
 const newX = ...; // calculate from e.clientX etc.  
 const newY = ...;  
 if (!frame) {  
 frame = requestAnimationFrame(() => {  
 useStore.getState().setPosition(newX, newY); // update Zustand store  
 frame = 0;  
 });  
 }  
};

This way, even if onDragMove runs many times, it will schedule at most one state update per frame. The **browser’s frame rate** dictates the update frequency, which is ideal for smooth rendering[[9]](https://nolanlawson.com/2019/08/11/high-performance-input-handling-on-the-web/#:~:text=The%20problem%20with%20Lodash%E2%80%99s%20,requestAnimationFrame)[[8]](https://nolanlawson.com/2019/08/11/high-performance-input-handling-on-the-web/#:~:text=Again%2C%20let%E2%80%99s%20take%20a%20step,one%20we%20need%20to%20paint). It prevents situations where multiple state updates queuing up cause layout thrash or slow frames. Throttling to requestAnimationFrame aligns updates with the natural paint cycle, improving drag performance.

Alternatively, a simple time-based throttle (e.g. update at most every 16ms) can also work. The key is to avoid calling set({x, y}) **for every single pixel movement**. Test different throttle intervals if needed – slower devices might benefit from a slightly lower frequency than 60 FPS. The result should be a balance where the dragged element tracks the cursor closely, but React isn’t doing more work than necessary.

## Consider Debouncing for Commit or Heavy Work

While throttling is great for continuous alignment, **debouncing** can be useful for deferring expensive computations until the drag is done (or paused). If updating the position itself is cheap but you have other side-effects (like saving the position, heavy collision checks, etc.), you might debounce those. For instance, you might update the Zustand store at a lower frequency or only on drag end, while using local state or refs for the real-time motion. Zustand’s maintainers suggest one approach: subscribe to the store and update a local piece of state with a debounced callback[[10]](https://github.com/pmndrs/zustand/discussions/1179#:~:text=I%20think%20you%20can%20subscribe,set%20local%20state%20with%20debounce). This way, your component’s render only fires at the debounced rate, not every rapid update.

In practice, debouncing means *waiting* until the user stops moving (or slowing down) before triggering an update. This can prevent “flurries” of updates from causing a cascade of re-renders. For example, if you only need the final position in global state (for persistence or future renders), you could hold the live position in a ref during the drag and call setPosition in the store when the drag ends (on pointerup). The UI element still moves (because you update it via ref or local state), and then one final store update occurs – thus no reactive updates are lost, but intermediate re-renders were avoided entirely. This pattern keeps your app responsive under rapid drag movements.

However, if the position *is* needed in the store continuously (perhaps other components read it in real time), combining a mild throttle with selective subscriptions is usually preferable to a long debounce. Debounce is best for things like window resizing or finalizing an operation, whereas throttle (or rAF) is better for steady feedback during an ongoing action like dragging[[9]](https://nolanlawson.com/2019/08/11/high-performance-input-handling-on-the-web/#:~:text=The%20problem%20with%20Lodash%E2%80%99s%20,requestAnimationFrame).

## Leverage Transient Updates with Refs (No Re-render on Every Move)

Zustand provides a mechanism for **transient state updates** – capturing state changes without forcing React to reconcile each time[[11]](https://github.com/pmndrs/zustand#transient-updates-for-often-occurring-state-changes#:~:text=Transient%20updates%20,changes). This is perfect for drag operations. The idea is to use the store’s low-level subscription to update something like a ref or an effect, instead of always updating via React state. For example, you can subscribe to the store’s coordinates and assign them to a useRef value on each change:

const posRef = useRef(useStore.getState().position);  
useEffect(() => {  
 // subscribe to updates without causing component re-renders  
 const unsub = useStore.subscribe(  
 state => { posRef.current = state.position; }  
 );  
 return unsub;  
}, []);

In this setup, posRef.current will always have the latest {x, y} from the store, but your component isn’t re-rendering on each update[[12]](https://github.com/pmndrs/zustand#transient-updates-for-often-occurring-state-changes#:~:text=The%20subscribe%20function%20allows%20components,to%20mutate%20the%20view%20directly). You can then use posRef.current inside a requestAnimationFrame loop or in an animation library to move the element. In fact, you can go a step further: update the DOM directly in the subscription callback. For instance, if you have a ref to the DOM node of the draggable element, you could do:

useEffect(() => {  
 const unsub = useStore.subscribe(state => {  
 myElementRef.current.style.transform = `translate(${state.x}px, ${state.y}px)`;  
 });  
 return unsub;  
}, []);

This imperatively moves the element on each state change without React’s involvement, leading to drastic performance gains in high-frequency scenarios[[12]](https://github.com/pmndrs/zustand#transient-updates-for-often-occurring-state-changes#:~:text=The%20subscribe%20function%20allows%20components,to%20mutate%20the%20view%20directly). The official Zustand docs note that binding a component to store changes via subscribe (and cleaning up on unmount) can **“make a drastic performance impact when you are allowed to mutate the view directly.”**[[11]](https://github.com/pmndrs/zustand#transient-updates-for-often-occurring-state-changes#:~:text=Transient%20updates%20,changes) In other words, by skipping React renders and manipulating the element, you avoid any chance of React causing lag or breaking the pointer interaction mid-drag.

Use this pattern judiciously: you’ll need to ensure the UI remains in sync. Typically, this is used for very frequent, ephemeral updates (like drag position or canvas drawing coordinates) that don’t need to trigger full component re-renders each time. The state is still stored in Zustand (so other subscribers or future renders can get it), but you’re handling the visual update imperatively for smoothness. This approach was even recommended by Zustand’s maintainers for cases of **often-occurring state changes**[[10]](https://github.com/pmndrs/zustand/discussions/1179#:~:text=I%20think%20you%20can%20subscribe,set%20local%20state%20with%20debounce). They suggest subscribing and using a ref or local state updated with a debounced handler as needed, rather than directly binding a component’s render to a highly volatile value.

**Important:** If you go this route, be mindful of *reactivity*: since you bypass React updates, other components depending on those values won’t update automatically unless they also subscribe or use the store. This is usually fine if the dragging is self-contained. If another part of your UI (say a coordinates display) needs updates, you can subscribe there as well or use a throttled state update to inform it.

## Avoid Circular Updates and Preserve Drag Interaction

When updating state during drags, it’s possible to introduce feedback loops or lose the drag “focus” if not handled properly. Here are some tips to avoid that:

* **Don’t update state in response to itself in a loop:** For example, avoid a pattern where a useEffect listens to store changes and calls setState again immediately – this can cause an update loop. Only trigger state changes from user interactions or purposeful events, not directly from the state updates unless absolutely necessary (and if so, use conditions to break cycles).
* **Keep transient calculation data out of Zustand:** Often during a drag, you need to store the initial grab point, offsets, or other info just for calculating the new position. Store these in useRef or component-local state instead of the global store. They don’t need to be global or reactive. For instance, you might record dragStartX/Y and element’s initial coordinates in refs on pointer down. These can be used to compute new x, y on pointer move without any extra global state updates. This avoids unnecessary store writes and re-renders while dragging, and also prevents issues where intermediate values cause state changes that could conflict with the next pointer event.
* **Maintain pointer capture:** React re-renders (if they happen) should ideally not disrupt the drag. Make sure the draggable element is not being unmounted or drastically changed during the drag. Using the pointer events API, call e.currentTarget.setPointerCapture(e.pointerId) on pointer down to keep receiving pointer events even if the pointer leaves the element. As long as the DOM node remains, pointer capture ensures you won’t “lose” the drag if React does a re-render on a state update. In practice, if you follow the above advice (throttle updates and avoid heavy changes), the element will stay in place and continue to receive events. But pointer capture is a good safety measure to prevent losing events mid-drag due to any re-render or the pointer moving fast outside the element.
* **Avoid key resets or remounts**: Ensure the element’s React key remains stable so that React doesn’t replace the component during updates. A remount would drop any pointer capture and event listeners, effectively interrupting the drag. Keep the structure of your component such that the draggable item is persistent while dragging.

If you find that even throttled state updates are causing slight hiccups in the drag (maybe due to very heavy sibling updates or other computations), consider reducing the frequency further or using the aforementioned ref subscription approach to completely decouple the visual movement from React’s lifecycle. The goal is that **React’s reconciliation should not interfere with the pointer interaction**. Zustand’s author has noted that if some state is not needed for rendering (i.e. non-reactive), you might not even put it in the store at all[[13]](https://github.com/pmndrs/zustand/discussions/2886#:~:text=Zustand%2C%20like%20React%2C%20expects%20immutable,the%20context%20of%20my%20requirements)[[14]](https://github.com/pmndrs/zustand/discussions/2886#:~:text=Answered%20by%20%20dai,63). In a discussion about high-frequency pointer updates, it was suggested that keeping such data in a mutable structure (like a closure or ref) and only exposing minimal derived state (if any) to React is fine[[15]](https://github.com/pmndrs/zustand/discussions/2886#:~:text=1,structures%20would%20significantly%20improve%20performance)[[14]](https://github.com/pmndrs/zustand/discussions/2886#:~:text=Answered%20by%20%20dai,63). This way, you never risk a “circular” update where updating the store triggers a re-render that tries to update the store again.

## Example: Store and Component Setup

To put it all together, here’s a conceptual example of how you might structure a Zustand store and a draggable component:

// Zustand store  
const useDragStore = create((set) => ({  
 x: 0,  
 y: 0,  
 setPosition: (x, y) => set(state => {  
 // Only update if values actually changed (to avoid extra re-renders)  
 if (state.x !== x || state.y !== y) {  
 return { x, y };  
 }  
 return state; // no change  
 })  
}));

In the component:

function DraggableItem() {  
 // subscribe to x and y; component only re-renders when these change  
 const [x, y] = useDragStore(state => [state.x, state.y]);  
 const itemRef = useRef(null);  
  
 // On mount, set up an imperative subscription for smooth movement (optional)  
 useEffect(() => {  
 const unsub = useDragStore.subscribe(state => {  
 // Directly apply transform for immediate response  
 if (itemRef.current) {  
 itemRef.current.style.transform = `translate(${state.x}px, ${state.y}px)`;  
 }  
 });  
 return unsub;  
 }, []);  
  
 // Pointer event handlers  
 const startPosRef = useRef({ offsetX: 0, offsetY: 0 });  
 const handlePointerDown = (e) => {  
 // capture pointer and record offset of cursor within the element  
 e.currentTarget.setPointerCapture(e.pointerId);  
 const rect = e.currentTarget.getBoundingClientRect();  
 startPosRef.current.offsetX = e.clientX - rect.left;  
 startPosRef.current.offsetY = e.clientY - rect.top;  
 };  
 const handlePointerMove = (e) => {  
 if (e.currentTarget.hasPointerCapture(e.pointerId)) {  
 // Compute new position (e.g., ensure the element moves with cursor minus initial offset)  
 const newX = e.clientX - startPosRef.current.offsetX;  
 const newY = e.clientY - startPosRef.current.offsetY;  
 // Throttle this update – e.g., only set state if enough time passed  
 useDragStore.getState().setPosition(newX, newY);  
 }  
 };  
 const handlePointerUp = (e) => {  
 e.currentTarget.releasePointerCapture(e.pointerId);  
 // (Possibly trigger a final state update or any onDragEnd logic here)  
 };  
  
 return (  
 <div   
 ref={itemRef}  
 onPointerDown={handlePointerDown}  
 onPointerMove={handlePointerMove}  
 onPointerUp={handlePointerUp}  
 style={{  
 position: 'absolute',  
 transform: `translate(${x}px, ${y}px)`  
 }}  
 >  
 Draggable Item  
 </div>  
 );  
}

In this setup, the component uses the **store’s reactive state** (x and y) to set its initial position and to re-render if the position changes due to other reasons (e.g., programmatic updates or another device). But during an active drag, we also imperatively update the element’s style for snappy feedback. The state is updated via setPosition (which is throttled in this example by only updating when values actually change; you could integrate a requestAnimationFrame check in handlePointerMove for finer control). The use of pointer capture and refs for offsets ensures that we don’t lose track of the drag and don’t put those intermediate calculations into global state.

This hybrid approach ensures **no loss of reactivity** – the state in the store does eventually reflect the final position (and any other interested components can get updates), but we avoid flooding React’s rendering pipeline with updates at the drag’s peak frequency. The result is a smooth drag with maintained interactivity.

## References and Further Reading

* **Zustand Official Docs – Transient Updates:** Demonstrates subscribing to store changes (e.g. a rapidly updating “scratch” count) and updating a ref instead of re-rendering, which greatly improves performance for frequent updates[[11]](https://github.com/pmndrs/zustand#transient-updates-for-often-occurring-state-changes#:~:text=Transient%20updates%20,changes).
* **Zustand GitHub Q&A (#1179):** Recommendation by a maintainer to handle rapid updates by subscribing to the store and updating local state with a debounce/throttle, referencing the transient updates pattern[[10]](https://github.com/pmndrs/zustand/discussions/1179#:~:text=I%20think%20you%20can%20subscribe,set%20local%20state%20with%20debounce).
* **Zustand GitHub Q&A (#2886):** Discussion of storing non-reactive state (like high-frequency pointer positions) in Zustand. The maintainer suggests using transient updates or even not putting such data in the store if it isn’t needed for rendering[[13]](https://github.com/pmndrs/zustand/discussions/2886#:~:text=Zustand%2C%20like%20React%2C%20expects%20immutable,the%20context%20of%20my%20requirements)[[14]](https://github.com/pmndrs/zustand/discussions/2886#:~:text=Answered%20by%20%20dai,63).
* **TkDodo’s Blog – Working with Zustand:** Tips on structuring Zustand state, including using selectors for individual values to avoid unwanted re-renders across the app[[1]](https://tkdodo.eu/blog/working-with-zustand#:~:text=2%20const%20,useBearStore). This reinforces avoiding subscription to the entire store and keeping selectors stable (e.g., not returning new objects)[[2]](https://tkdodo.eu/blog/working-with-zustand#:~:text=Effectively%2C%20this%20means%20that%20selectors,the%20content%20is%20the%20same).
* **React Flow Performance Guide (Synergy Codes):** Highlights performance pitfalls when state updates cause many components to re-render during drag. Recommends isolating frequently changing state and using Zustand’s selectors (with shallow equality) to prevent cascading re-renders[[5]](https://www.synergycodes.com/blog/guide-to-optimize-react-flow-project-performance#:~:text=The%20main%20issue%20arises%20from,are%20being%20dragged%20or%20not)[[16]](https://www.synergycodes.com/blog/guide-to-optimize-react-flow-project-performance#:~:text=An%20alternative%20approach).
* **Nolan Lawson – High-performance Input Handling:** (General concept) Explains why events like pointermove should be throttled to the browser’s frame rate to avoid doing redundant work between repaints[[8]](https://nolanlawson.com/2019/08/11/high-performance-input-handling-on-the-web/#:~:text=Again%2C%20let%E2%80%99s%20take%20a%20step,one%20we%20need%20to%20paint). This concept underpins using requestAnimationFrame or similar throttling for drag updates.

By following these practices – using fine-grained selectors, throttling updates, leveraging refs for transient values, and structuring your store thoughtfully – you can maintain **both** React’s reactivity and a smooth drag experience. The goal is to update state in a way that React can keep up with, without dropping frames or losing the handle on the drag interaction.

[[1]](https://tkdodo.eu/blog/working-with-zustand#:~:text=2%20const%20,useBearStore) [[2]](https://tkdodo.eu/blog/working-with-zustand#:~:text=Effectively%2C%20this%20means%20that%20selectors,the%20content%20is%20the%20same) Working with Zustand | TkDodo's blog

<https://tkdodo.eu/blog/working-with-zustand>

[[3]](https://philipp-raab.medium.com/zustand-state-management-a-performance-booster-with-some-pitfalls-071c4cbee17a#:~:text=2,directly%20affected%20by%20that%20state) [[4]](https://philipp-raab.medium.com/zustand-state-management-a-performance-booster-with-some-pitfalls-071c4cbee17a#:~:text=to%20a%20set%20of%20listeners,and%20subsequently%20calls%20all%20listeners) Zustand state management: A performance booster with some pitfalls | by Philipp Raab | Medium

<https://philipp-raab.medium.com/zustand-state-management-a-performance-booster-with-some-pitfalls-071c4cbee17a>

[[5]](https://www.synergycodes.com/blog/guide-to-optimize-react-flow-project-performance#:~:text=The%20main%20issue%20arises%20from,are%20being%20dragged%20or%20not) [[16]](https://www.synergycodes.com/blog/guide-to-optimize-react-flow-project-performance#:~:text=An%20alternative%20approach) Synergy Codes — The ultimate guide to optimizing React Flow project performance [EBOOK]

<https://www.synergycodes.com/blog/guide-to-optimize-react-flow-project-performance>

[[6]](https://nolanlawson.com/2019/08/11/high-performance-input-handling-on-the-web/#:~:text=There%20is%20a%20class%20of,the%20browser%20can%20paint%20frames) [[7]](https://nolanlawson.com/2019/08/11/high-performance-input-handling-on-the-web/#:~:text=Image%3A%20Screenshot%20of%20Chrome%20Dev,many%20as%20four%20pointermove%20events) [[8]](https://nolanlawson.com/2019/08/11/high-performance-input-handling-on-the-web/#:~:text=Again%2C%20let%E2%80%99s%20take%20a%20step,one%20we%20need%20to%20paint) [[9]](https://nolanlawson.com/2019/08/11/high-performance-input-handling-on-the-web/#:~:text=The%20problem%20with%20Lodash%E2%80%99s%20,requestAnimationFrame) High-performance input handling on the web | Read the Tea Leaves

<https://nolanlawson.com/2019/08/11/high-performance-input-handling-on-the-web/>

[[10]](https://github.com/pmndrs/zustand/discussions/1179#:~:text=I%20think%20you%20can%20subscribe,set%20local%20state%20with%20debounce) Debouncing state slices · pmndrs zustand · Discussion #1179 · GitHub

<https://github.com/pmndrs/zustand/discussions/1179>

[[11]](https://github.com/pmndrs/zustand#transient-updates-for-often-occurring-state-changes#:~:text=Transient%20updates%20,changes) [[12]](https://github.com/pmndrs/zustand#transient-updates-for-often-occurring-state-changes#:~:text=The%20subscribe%20function%20allows%20components,to%20mutate%20the%20view%20directly) GitHub - pmndrs/zustand: Bear necessities for state management in React

<https://github.com/pmndrs/zustand>

[[13]](https://github.com/pmndrs/zustand/discussions/2886#:~:text=Zustand%2C%20like%20React%2C%20expects%20immutable,the%20context%20of%20my%20requirements) [[14]](https://github.com/pmndrs/zustand/discussions/2886#:~:text=Answered%20by%20%20dai,63) [[15]](https://github.com/pmndrs/zustand/discussions/2886#:~:text=1,structures%20would%20significantly%20improve%20performance) Best Practices for Storing Non-Reactive State in Zustand · pmndrs zustand · Discussion #2886 · GitHub

<https://github.com/pmndrs/zustand/discussions/2886>