ChatGPT

Ring UI System Implementation Plan

This document outlines a comprehensive plan to implement the **Ring (Pergamino) UI scrolling system** for the Portafolio_v02 project. The plan is organized by feature, aligning with the Rings UX Blueprint and adhering to the project's stack (React, Vite, TypeScript, TailwindCSS, shadcn/ui) and professional baseline guidelines. Each section below describes the feature goal, the files/modules involved (based on the proposed structure in the blueprint), and a detailed checklist of development tasks. The focus is on maintainability, clean architecture separation, logging hooks for insight, and modular components.

Ring Data Model & Configuration

Goal: Establish a declarative configuration for all rings and their items, and define data models to drive the ring system. This provides a single source of truth for ring parameters (angles, spacing, snap behavior, etc.) and content, enabling easy tweaks without hardcoding. It also lays the groundwork for a global UI state to track active ring and modes.

Files/Modules to Create or Update:

- `src/rings/config/rings.json` – JSON specification of ring order, each ring's settings (axis angle, spacing, snap tolerances, etc.), and its items [1] [2]. This will list the initial five rings (e.g. `"landing", "work", "about", "contact", "settings"`) and at least a couple of items per ring as placeholders/content.
- **Type Definitions** (e.g. in a `types.ts` or within RingEngine module) – Define TypeScript interfaces/types for `Ring`, `RingItem`, and related structures as described in the blueprint [3] [4] (id, name, axisDeg, items array, etc., including nested `BackgroundSpec`). These ensure the config JSON is strongly typed.
- **Global State** – Determine where to manage global UI state (e.g., a React Context or Zustand store) to hold `uiMode` (desktop/mobile), `activeRingId`, `isAltFrameActive`, and `isNavOpen` [5]. This may involve creating a context provider (e.g. `src/rings/RingsProvider.tsx`) or integrating into an existing top-level state (like App).

Development Tasks:

- [ ] **Create Rings Config:** Draft the `rings.json` file with the initial ring data (using the blueprint's example as a template [2]). Include five ring entries with properties like `axisDeg`, `spacingPx`, `snapTolerance`, `idleSnapDelayMs`, `background` (color/gradient config), and a few `items` (placeholders or sample content).
- [ ] **Define Data Interfaces:** Write TypeScript interfaces for the config structure (`Ring`, `RingItem`, `BackgroundSpec`, etc.) in a suitable location. Ensure fields match the JSON (e.g., `axisDeg: number`, `loop: boolean`, `spacingPx: number`, etc. from the blueprint model) [3]. Include derived runtime state fields (like `ringIndex`, `ringIndexProgress`, `velocity`, `lastInputAt`) either in the RingEngine or as part of a Ring state type [6].
- [ ] **Global State Setup:** Implement a global UI state management solution:
- If using React Context: create a `RingsProvider` with a reducer or useState to hold `activeRingId`, `uiMode`, etc., and provide methods to change rings and toggle alt frames.
- Initialize `activeRingId` to the first ring (from `ringOrder` in config) on app load.

- Determine `uiMode` by checking screen width or user agent and update on resize (or use CSS media queries to conditionally render desktop vs mobile nav).
- [ ] **Load Config Data:** On app startup, load or import `rings.json`. Since this is a Vite app, we can import JSON directly as an object. Validate it against the TypeScript types (or use `as RingConfig` assertion) so that any mismatch is caught at compile time.
- [ ] **Ensure Clean Architecture:** Keep configuration and state logic separate from UI rendering. For example, parsing `rings.json` and setting up data should be done in the provider or a setup function, not sprinkled in components, to adhere to modular design.
- [ ] **Logging Hooks for Config:** If needed, log (in development) that the rings config was loaded and how many rings/items to verify everything is read correctly. (This can tie into the debug panel later.)

Ring Engine (Continuous Loop Mechanism)

Goal: Implement the core ring physics and state management. The RingEngine is responsible for treating a list of items as an endless loop, updating the ring's scroll position (continuous index), and wrapping around seamlessly. It provides the mechanism to advance or reverse the ring and calculates derived values like progress and velocity, independent of the UI. This encapsulated logic ensures maintainability and testability of the scroll behavior.

Files/Modules to Create or Update:

- `src/rings/engine/RingEngine.ts` – Core logic for ring state, scrolling, and wrapping behavior [7]. This will be a TypeScript class or module without direct DOM dependencies. It holds the ring's length (N items), current `ringIndex` (which can be fractional for positions between items), and methods to update this index.
- (Potentially) `src/rings/engine/types.ts` – If not already defined elsewhere, include the `Ring` and `RingItem` type definitions or import them here.
- **Integration in Component** – The `RingEngine` will be used by the RingView component (but the integration is handled in the RingView section). Ensure the engine can emit events or callbacks for state changes if needed.

Development Tasks:

- [ ] **Implement RingEngine Class:** Create the RingEngine with a constructor that takes a `Ring` config object (or necessary params like number of items, axis angle, etc.). Initialize state properties:
- `ringIndex` (continuous index, start at 0).
- `ringIndexProgress` (normalized 0–1 progress = `ringIndex / N` for convenience) [6].
- `velocity` (pixels per second or similar, initially 0).
- `lastInputAt` (timestamp of last input, for idle tracking).
- Store any config like `spacingPx`, `axisDeg`, `snapTolerance`, etc., for internal use.
- [ ] **Continuous Wrapping Logic:** Implement a method (e.g. `updateIndex(delta: number)`) to advance the ring position by a delta (which comes from input):
- Update `ringIndex += delta`.
- Wrap the `ringIndex` within `[0, N)` to ensure it loops endlessly (e.g., using modulo: `ringIndex = (ringIndex % N + N) % N`) [8].
- Update `ringIndexProgress = ringIndex / N`.
- Optionally, compute `velocity` based on delta and time (you can track time between updates to derive a velocity if needed for snap animations).

- [ ] **Axis Consideration:** The engine can be agnostic of the visual axis rotation; it mainly cares about the index. The `axisDeg` from config will be used in the rendering layer to orient the movement. However, note the axis for potential future use (like certain physics tweaks if needed).
- [ ] **Loop Seamlessness:** Ensure that when `ringIndex` wraps around, the transition is seamless. This might involve logic to handle negative deltas properly (using modulo as above). We must guarantee **no perceptible seam** at the boundaries [9].
- [ ] **Expose Controls:** Provide methods such as `getIndex()`, `setIndex()`, and maybe `setVelocity()` or `dampenVelocity()`. These will be useful for integrating snapping (e.g., an external Snapper might call `engine.setIndex(target)` gradually). Keep methods small and purpose-driven.
- [ ] **Velocity Tracking:** Implement basic velocity calculation. For example, each time `updateIndex` is called, set `lastInputAt = Date.now()` and compute velocity = `deltaDistance / deltaTime` (with appropriate units). Cap the velocity if it exceeds a threshold (to avoid nausea from extremely fast scrolls) [10].
- [ ] **Maintainability:** Keep this logic free of UI concerns (no direct DOM). Add comments (TSDoc) explaining the algorithm and any formulas (like how wrapping works). Include unit tests or at least plan to manually test edge cases: e.g., advancing more than N in one go should still wrap correctly.
- [ ] **Logging Hooks:** Integrate hooks from the baseline: whenever `ringIndex` changes, if a callback `onIndexChange` is registered, call it with current ringId, index, and velocity [11]. This will later feed the debug panel. Ensure these callbacks (if any) are optional and dev-only.

Input Handling (Desktop & Mobile Controls)

Goal: Map user input into ring movement in a natural way. On desktop, the primary control is the mouse wheel (vertical scroll) – this should advance the ring along its configured axis (which may be horizontal, vertical, or angled). On mobile, touch gestures (vertical swipes) serve the same purpose. This feature covers creating an InputMapper that translates raw input events into calls to the RingEngine, handling differences between devices, and ensuring a smooth experience (including special actions like double-tap on mobile to switch rings).

Files/Modules to Create or Update:

- `src/rings/engine/InputMapper.ts` – Module to handle input events and convert them to ring motion [7]. It will likely export functions or a class that can be initialized with references to a RingEngine (or a way to dispatch updates to it).
- **Integration in Components:** The RingView component (or higher-level App) will utilize InputMapper:
- For desktop: attach wheel event listener.
- For mobile: attach touch/start/move/end listeners.
- For mobile dock: handle double-tap (this might be handled in the Dock component rather than InputMapper directly).
- **Global Settings:** If there's a need for a global invert scrolling or sensitivity setting, InputMapper should respect it (e.g., an option to invert delta sign) [10].

Development Tasks:

- [ ] **Desktop Wheel to Scroll:** Inside InputMapper, implement logic to handle the `wheel` event:
- Listen for `wheel` events on the ring container element (RingView will attach it via React, e.g., `onWheel`).

- Read `event.deltaY` – this represents vertical scroll. Compute a movement delta for the ring: `Δ = k * deltaY`, where `k` is a sensitivity constant (tune experimentally or derive from spacing) [8].
- Map this delta to the ring's axis: since the ring's scroll axis could be angled (axisDeg), project the delta onto that axis. (One approach: rotate the delta vector by `axisDeg` to get X/Y movement, but since deltaY is vertical input, a simpler interpretation is to treat `deltaY` as movement along whatever axis is defined. We can multiply `deltaY` by cos(axisDeg) for horizontal component and sin(axisDeg) for vertical if needed. Alternatively, rotate the container as in RingView and use delta directly.)
- Call `RingEngine.updateIndex(Δ_converted)` to update the ring's position.
- Prevent default scrolling of the page if appropriate, so the canvas doesn't cause whole-page scroll.
- [ ] **Mobile Touch to Scroll:** Implement touch event handling:
- Attach `onTouchStart` and `onTouchMove` on the ring container (or possibly use a specialized hook like useGesture, but manual is fine).
- Track the touch's Y movement (we restrict to vertical swiping only as per spec) [12]. Calculate `deltaY` from touch start to current touch move.
- Convert that to a ring movement delta (similarly to wheel, possibly the same `k` sensitivity).
- Call `RingEngine.updateIndex(Δ)` accordingly during the move (for smooth real-time movement). For performance, use `requestAnimationFrame` or throttling if needed to avoid overly frequent updates on fast swipes.
- On touch end, you might not need to handle inertia manually at first (no flick velocity carry-over unless we explicitly want to add momentum – not specified, so initially we can stop movement when touch ends).
- [ ] **Dock Double-Tap (Mobile):** In the Dock component, implement logic to switch rings on a double tap [13]:
- Use a state/timer to detect double taps on a specific area (e.g., the dock icon or the whole dock). If two taps occur within a short time window, interpret as double-tap.
- On double-tap, invoke the ring change to the next ring in order (e.g., find current ring's index in ringOrder array and increment with wrap-around). This will use the ring switching logic defined later (via `changeRing()` function).
- Provide visual feedback if needed (maybe a small flash or icon change to indicate the action).
- [ ] **Integrate InputMapper with Engine:** Ensure the InputMapper has access to the correct RingEngine instance (for the active ring). When the active ring changes, InputMapper should update to control the new ring's engine. This might mean:
- Either one global InputMapper that calls into the active ring's engine (determined via `activeRingId` and a mapping of ringId -> engine).
- Or reinitialize/retarget the event handlers when active ring changes.
- Decide approach for simplicity: e.g., have a dictionary of ring engines, and InputMapper functions take an engine reference as argument each call.
- [ ] **Smooth Movement & Tuning:** Fine-tune the sensitivity `k` so that scrolling feels natural. Possibly make it relative to `spacingPx` (so one notch of wheel moves ~some fraction of an item). Check that small wheel motions produce slight movement and larger/faster scrolls can spin multiple items.
- [ ] **Optional Enhancements:** Consider implementing **inertia** for touch (if a fast swipe should continue moving a bit). This can be done by capturing velocity on touch end and continuing to call updateIndex with decaying velocity (could integrate GSAP's inertial plugin or manually via `requestAnimationFrame`). This is not strictly in the initial spec, but could improve feel.
- [ ] **Edge Settings:** Read any global settings like invert scroll direction or velocity clamp:
- If a config or user setting indicates inverted scrolling, multiply delta by -1 before applying [10].
- Clamp extremely large `deltaY` values or velocity to avoid very sudden jumps.

- [ ] **Testing Input:** Manually test on desktop: scrolling down should advance the ring in the intended direction (e.g., for a horizontal ring, maybe scrolling down moves items left/right depending on design). Ensure no horizontal panning with pointer is needed (wheel only) [14].
- On mobile emulator or device: swipe up/down moves the ring appropriately (no horizontal drag). Test that a quick double-tap on the dock does switch rings [14].
- [ ] **Accessibility Note:** While implementing, ensure these event handlers don't block or interfere with assistive technology (we'll add explicit keyboard handlers later). E.g., wheel and touch are fine; just be mindful if any pointer events are canceled.
- [ ] **Logging:** Each input event that causes a ringIndex change will naturally trigger the `onIndexChange` hook (from RingEngine). Additionally, perhaps log raw input deltas in dev mode if investigating touch vs wheel behavior differences.

Idle Snap-to-Midpoint Behavior

Goal: Implement the **idle snapping** feature, where after a short period of no user input, the ring automatically eases to a neat alignment. Specifically, it should snap to the **midpoint** between items (halfway through an item's scroll length) by default, or to an exact item center if the current position is very close to one. This ensures the ring doesn't stop in an awkward in-between state and improves the visual polish. The Snapper logic will monitor inactivity and handle the smooth animation to the target index.

Files/Modules to Create or Update:

- `src/rings/engine/Snapper.ts` – Module implementing the snap algorithm [15]. It might export a function or class that can be given a RingEngine (or current ring state) and perform the snapping.
- **Integration** – The Snapper will likely be used within the RingEngine or managed by the global ring controller. We might integrate it by calling Snapper's check on an interval or using `requestAnimationFrame`. Possibly, Snapper could also be a small utility inside RingEngine for simplicity.
- `RingEngine.ts` – may need modifications to cooperate with Snapper (like exposing current index, setting index, etc., which we already plan to have).

Development Tasks:

- [ ] **Implement Snapper Logic:** Following the blueprint's algorithm [16], implement a function `computeSnapTarget(currentIndex, N, tolerance)`:
- Let `i0 = floor(currentIndex)` (the index of the item currently at or just before the viewport center).
- Let `t = currentIndex - i0` (the fractional part, i.e., how far into the next item we are).
- If `t` is approximately 0.5 (within a ±tolerance range of the midpoint 0.5) → already near the midpoint, so do nothing (no snap needed) [17].
- Else compute midpoint index `m = i0 + 0.5` (mod N, meaning 0.5 past item i0, which is the midpoint between i0 and i0+1).
- Determine **center vs midpoint**: if the current position is very close to a true item center (tolerance defined by `snapTolerance` fraction of spacing), then snapping to that center might look better. In other words, if `t` is very small or very close to 1.0 (within tolerance of 0 or 1), choose the nearest item index (i0 or i0+1) as target instead of the midpoint [18].
- Otherwise, target the midpoint `m`.
- [ ] **Idle Timer:** Decide how to detect idle time:

- We can use `RingEngine.lastInputAt`. Set up a repeated check (e.g., `setInterval` or an `requestAnimationFrame` loop) that computes `time since last input`.
- Alternatively, use `setTimeout` that resets on each input. For example, on each wheel/touch event, clear any existing timeout and start a new one for `idleSnapDelayMs`.
- Using a timeout might be simpler: on input, reset a timer. When the timer fires (no input for X ms), trigger the snap.
- [ ] **Integrate with RingEngine:** Implement a method like `RingEngine.maybeSnap()` or handle externally:
- Possibly have Snapper as an internal part of RingEngine: the engine's updateIndex could schedule the snap timeout.
- Or have a separate SnapManager that lives at a higher scope and monitors the active ring's engine.
- Simpler: inside RingEngine, when updating `lastInputAt`, also handle the timer logic (with access to `idleSnapDelayMs` from config).
- [ ] **Animate the Snap:** Use a smooth animation to move from current `ringIndex` to the target:
- Because this involves interpolation, leverage either CSS transitions, Framer Motion, or GSAP. For a straightforward approach, we might directly manipulate the ringIndex over time.
- If using GSAP, we can tween the numeric `ringIndex` value to the target, with an easing like `expo.out` or a spring for critical damping [19]. GSAP can call a callback on each tick to update the engine's index.
- If using Framer Motion, we could treat `ringIndex` as a Motion Value and use `.animate` to a target.
- Make sure to account for wrap-around (if target index is lower than current because of modulo wrap, you might adjust math to ensure shortest path, but since ringIndex is continuous beyond N, one can always choose a representation of target within a few of currentIndex).
- [ ] **Prevent Interruption Issues:** If the user provides input during an ongoing snap animation, cancel the snap and let user control take over. This means the input event should clear any ongoing tweens/animations.
- If using GSAP, kill the tween on new input.
- If using a custom RAF loop for snap, set a flag and break out if new input comes.
- [ ] **Snap Tuning:** Use `snapTolerance` and `idleSnapDelayMs` from the ring's config (loaded from JSON) to drive the behavior:
- E.g., default delay 900ms on landing ring (from blueprint) [20].
- Test different tolerances – if too high, it will snap to center even when not intended; if too low, it always goes midpoint except nearly dead center cases.
- [ ] **Visual Outcome:** The result should be that after the user stops interacting, within ~1 second the ring smoothly slides such that an item or midpoint is perfectly centered on screen. This should be subtle and feel "alive" rather than jarring.
- [ ] **Testing:** Manually test by stopping scroll at various positions:
- Halfway between items → should snap to midpoint exactly.
- Very close to an item → should magnet to that item's center (since within tolerance).
- Already near midpoint → should not move at all (stays as is).
- Ensure the snap animation is noticeable but not too slow (fast ease-out is suggested).
- Check that continuous minor adjustments (like tiny scrolls) reset the timer appropriately (so it doesn't snap while actively fiddling).
- [ ] **Logging Hook:** Implement `onIdleSnap(startIndex, targetIndex, mode)` hook to fire when a snap happens [21]. Mode can be `'midpoint'` or `'center'` depending on what target was chosen. Log this in the debug panel for developers to see snap behavior.
- [ ] **Accessibility Consideration:** If `prefers-reduced-motion` is on, consider either disabling the snap animation (jump straight to target) or using a quicker ease. The blueprint suggests increasing tolerance and reducing velocity for reduced motion [22], which effectively might cause

items to more readily snap or move slowly – we can incorporate that by checking a global reducedMotion flag:
- If reduced motion is true, perhaps skip animation and instantly set `ringIndex = target` when idle, to avoid any motion.

Ring Components (RingView & RingItem)

Goal: Build the React components that render the ring on screen using the data and engine. `RingView` is responsible for displaying a ring of items (cards or placeholders) in the correct layout and orientation, and updating their positions as the `ringIndex` changes. `RingItem` is a sub-component for each item, encapsulating its presentation (placeholder vs content). These components should ensure that the ring's visuals loop seamlessly and remain behind the navigation UI (with appropriate z-index and styling).

Files/Modules to Create or Update:

- `src/rings/components/RingView.tsx` – The main component for a ring's viewport [23]. It will consume the RingEngine (and possibly subscribe to its state) and the ring config (items array, etc.), and handle rendering of items in a circular/looping fashion.
- `src/rings/components/RingItem.tsx` – A component to render an individual item (either a placeholder card or actual content) [23].
- **Styles/Assets:** Likely will use TailwindCSS utilities for positioning and sizing. Also uses assets from `assets/rings/<ringId>/...` for images or placeholder graphics as needed [24].

Development Tasks:

- [ ] **Build RingItem component:**
- Accept props for the item data (`RingItem` interface: id, kind, payload, size).
- If `kind === 'placeholder'`: Render a placeholder card. For example, a `<div>` with a fixed size (use the provided `size.w` and `size.h`), a rounded border outline, and maybe a low-res image icon or patterned background indicating "loading/placeholder" (the blueprint suggests a lightweight `.webp` in the center) [25]. Use Tailwind to style (e.g., `border-2 border-dashed border-gray-500/50 rounded-xl` and a subtle background).
- If `kind === 'content'`: Render the actual content. This might be an image or interactive card. For now, it could be an `<img>` using `payload.src` (make sure the source is imported or available from the `assets` folder) or a stub component/text if content is not ready. Constrain it to the given size.
- Make the item content focusable if appropriate (e.g., add `tabIndex=0` on a wrapper so keyboard users can focus items).
- Ensure the component's outer container is positioned absolutely within the ring container (positioning will be handled by RingView).
- [ ] **Build RingView component:**
- It should take a `ringId` or `Ring` config as prop. On mount, it will likely instantiate or retrieve the corresponding RingEngine for that ring.
    - Possibly create a new RingEngine each time if rings are independent. Or have a central store of engines and pick by id.

- Render a container (e.g., a `div`) that acts as the **floating viewport** for the ring. This container likely is full-screen or a large area centered in the viewport (with margins) such that items can move within it [25]. We want items to disappear off edges and come back around.
  - Use Tailwind utility classes or style: e.g., `absolute inset-0 overflow-hidden` for a full area. But since items go behind nav, maybe the container should not cover the nav area to allow clicking nav? Alternatively, the nav is pointer-opaque above it anyway.
  - Ensure this container has a lower z-index than the nav (so that items go under the frosted nav). Use something like `z-0` for ring content and a higher z for nav.
- **Orientation and positioning:** We need to align items along the ring's axis (which could be horizontal, vertical or angled). Two approaches:
  1. *Transform method:* Set the container with a CSS transform `rotate(-axisDeg)` so that moving items horizontally in this rotated coordinate space actually moves them along the desired axis. Then we can position items along the X-axis (left-right) inside this rotated container.
  2. *Manual calculation:* Compute for each item an (x, y) position using `axisDeg`. For example, if axisDeg=30°, an item with index difference `d` from the active item center should be positioned at `x = d * spacingPx * cos(axis)`, `y = d * spacingPx * sin(axis)`. This gives more control if needed for partial visibility.
  3. The transform method is simpler: we can create an inner wrapper that is rotated, and translate items along X within it. We'll proceed with that approach for implementation simplicity.
- Inside the container, render each `RingItem`:
  - Determine the index offset of each item relative to the current `ringIndex`. For example, if `ringIndex = 2.3`, the item with actual index 2 is the one currently centered (with some offset), index 3 is next, etc.
  - Compute how far each item is from the center position in terms of spacing. A given item `i` should be positioned at `(i - ringIndex) * spacingPx` along the axis direction.
  - Because `ringIndex` can be fractional and wraps, use modular arithmetic to handle items near the edges. One trick: duplicate the list of items on both ends to ensure continuity when in between.
  - E.g., take items array and create a small extended array [last item, ...all items..., first item] so that at boundaries you have neighbor items available. Or mathematically, for each visible slot, compute an index = `(floor(ringIndex) + k) mod N` for some k around 0.
  - Position items: apply an inline style or Tailwind class for translation. For horizontal axis (0°), it's just `translateX(offsetPx)`. For a rotated axis, if we rotated the wrapper by -axisDeg, then we still translate in X of that wrapper.
  - Use `Framer Motion` or CSS transitions for smooth movement? Actually, since we handle movement continuously via updates to transform positions on each state change, React re-render might handle it but could be janky if very frequent. Instead:
  - Consider using a single React state for `ringIndex` and update it frequently (50+ fps) – might not be ideal for performance.
  - Alternatively, use `requestAnimationFrame` in a useEffect to smoothly step toward new positions (or rely on Snapper/GSAP for inertial motion).
  - As a simpler initial step, update state directly on wheel/touch events (which might be fine if events fire ~60fps or less). Optimize later if needed.
- Attach event handlers: connect the `onWheel`, `onTouchStart/Move` from InputMapper. Possibly, RingView can import InputMapper and call a setup function in a useEffect to bind

events to the container (using low-level event listeners for flexibility outside React synthetic events).
    - Or use React handlers and call InputMapper logic inside (e.g., onWheel={(e) => InputMapper.handleWheel(e, engine)}).
- Ensure to cleanup listeners on unmount (especially if using raw addEventListener).
- [ ] **Z-index and Layering:** Verify that the RingView content sits behind the nav UI:
- The nav (SidePanel/Dock) will have a higher z-index (we'll set those in their CSS). Possibly add a Tailwind class like `z-10` to nav and `z-0` to ring container.
- For the frosted effect to work, the nav likely has `backdrop-filter: blur()`. Items passing behind will be visible blurred. This requires the items to actually render underneath. Ensure the ring container does not use its own backdrop or weird stacking context that could interfere.
- [ ] **Responsive Behavior:** On different screen sizes, RingView might need to adjust margins or item scaling:
- For now, rely on the item sizes given. Possibly center the ring such that the currently centered item is roughly in view. If items are large, the ring container might overflow the viewport slightly; verify that's acceptable.
- The blueprint mentions generous default margins around the floating container [26]. We might apply padding/margin in CSS to ensure items have space to move off-screen.
- [ ] **Testing RingView:** Populate the `rings.json` with some dummy images or colored blocks to see the ring working. Manually scroll to see if items repeat seamlessly (e.g., going from last item back to first with no gap). Adjust the duplication logic if a gap is visible.
- [ ] **Focus Handling:** When the active ring changes or after a snap, ensure the focused element (if any) is an item in view. Possibly auto-focus the newly centered item's element (this ties into accessibility; see below for details) [27].
- [ ] **Documentation:** Add TSDoc comments on these components describing how they work (per professional baseline). For example, document props and the approach to positioning.
- [ ] **Tailwind & shadcn Integration:** Use Tailwind for layout and basic styling. If any UI primitive from shadcn/ui (like a ScrollArea or card component) can simplify styling, consider it – though the ring is quite custom, so likely we'll stick to custom divs. We might use Lucide icons for placeholders or nav, which is fine to note for later.

Navigation Panel & Dock (Ring Selection UI)

Goal: Develop the navigation UI that allows switching between different rings. On **desktop**, this is a side panel (likely a vertical list of section names or icons) that is always visible on the left, with a frosted glass background. On **mobile**, it's a dock at the bottom – a minimal floating bar that might show the current section and allows cycling through rings (via double-tap or maybe tapping icons if space permits). This feature ensures the user can change rings intentionally (since scrolling alone won't switch rings as per design [28]) and that the nav remains accessible at all times above the ring content.

Files/Modules to Create or Update:

- `src/nav/SidePanel.tsx` – The desktop side navigation component [29].
- `src/nav/Dock.tsx` – The mobile bottom dock component [29].
- `src/components/static/FrostedPanel.tsx` – A reusable styled container for frosted-glass panels [30] (could be used by SidePanel, Dock, and potentially AltFrame's mini-nav).
- Possibly `App.tsx` or main layout – to conditionally render SidePanel vs Dock based on `uiMode` and to include these in the interface.

Development Tasks:

- [ ] **Create FrostedPanel component:**
- This component wraps children in a stylized, semi-transparent panel. Use Tailwind to apply styles like `backdrop-blur-md bg-white/50 dark:bg-gray-800/50` (light translucent background for light mode, maybe a different tone for dark mode if needed).
- Add rounded corners or a subtle border as per aesthetic preference. Ensure it uses `pointer-events: auto` (so it can be interacted with) and is not too large to avoid blocking the whole screen.
- The FrostedPanel should accept a `className` or style prop for customization (so SidePanel can make it taller, Dock can make it wider and short).
- [ ] **Implement SidePanel (Desktop Nav):**
- Place the SidePanel as a fixed/floating element on the left side of the viewport, full height or near full height. Likely width could be ~60px to 80px if icons only, or larger if labels.
- Use the FrostedPanel as the container for visual style.
- Inside, render navigation items for each ring in `ringOrder` (from config). These could be:
    - Icons or abbreviations (maybe use Lucide icons matching each section if available, or simple text rotated vertically).
    - Alternatively, vertical text labels for each ring ("Landing", "Work", etc.). Ensure they are writing-mode vertical or simply rotated if narrow.
- Highlight the active ring's item (e.g., with a filled background or accent border).
- Attach onClick handler to each item: when clicked, call a handler to switch to that ring. This handler will use the global `changeRing(targetRingId, transition?)` function (to be implemented) to initiate the ring switch (with a transition like fade).
- Ensure the panel is always visible above the RingView: assign a high z-index (e.g., `z-20`). Also ensure it doesn't scroll with the content (use fixed positioning).
- Add a subtle hover effect for desktop UX (e.g., lighten the item on hover).
- (Optional) Include a toggle for any settings or an expandable section? If not needed, skip.
- [ ] **Implement Dock (Mobile Nav):**
- The Dock will appear at the bottom center or bottom of the screen. Use FrostedPanel for consistency (with horizontal orientation).
- Likely a thinner bar (say height 50-60px) that stretches a bit but not full width (to avoid covering too much content).
- Content of Dock: Because mobile screens are small, listing all rings might not fit. The blueprint implies **double-tap** to cycle rings [31], so the Dock might just show the current ring's name or icon, and rely on double-tap for switching.
    - Possibly show an icon or label for the current section, and maybe a small indicator for others (like dots or a single icon that indicates "tap again for next").
- Implement double-tap detection on the Dock area:
    - If a double-tap is detected, call `changeRing(nextRingId)` to advance to the next ring in order. Provide a brief feedback (maybe flash the panel or change text) to indicate a switch.
    - You might also handle a single tap differently (e.g., a single tap could open an overlay menu listing all sections, if desired, but that's beyond the current spec. The spec only explicitly mentions double-tap for cycling).
- Make sure the Dock is only visible in mobile view. Use `uiMode === 'mobile'` or a CSS media query (hidden on md+ screens for instance).
- Style: center-align the content, perhaps show an icon (maybe a home icon for landing, briefcase for work, etc., if using icons). Use the project's icon library (Lucide) for consistency if using icons.
- [ ] **Global Nav Integration:** In the main App or layout, conditionally render `<SidePanel />` or `<Dock />` based on the current `uiMode`. The `uiMode` can be determined via effect on

mount (check window.innerWidth or use a CSS approach). Also possibly adjust if window is resized beyond a threshold (though it's rare to resize on mobile, but on desktop narrowing could trigger mobile layout).

- [ ] **changeRing Functionality:** Implement a global function or context action `changeRing(targetRingId, transition?)` (referenced in blueprint as API) [32]:
- If called, this should update the global state `activeRingId` to `targetRingId` and handle any transition animation. We plan transitions in the next section; for now, ensure the basics:
- If no transition specified, or type `'cut'`, simply unmount current RingView and mount the new RingView for the target ring immediately.
- If transition is provided (e.g., `'fade'`), trigger the appropriate animation (fade current out, fade new in).
- The SidePanel onClick can call `changeRing(id, { type: 'fade', durationMs: 300 })` as an example.
- Make sure to disable or ignore additional `changeRing` calls if one is mid-transition (to avoid rapid clicking issues).
- [ ] **Nav Visibility & Behavior:** Ensure that:
- The nav panel is always on top of other content (test z-index over ring items).
- Ring items scroll under it (you should see them through the frosted glass as they pass behind) [33].
- The SidePanel does not scroll with wheel (so add `pointer-events: none` to it when the user scrolls the page? Alternatively, since wheel events on the nav should not do anything, possibly just let them bubble to RingView).
- For Dock, check that swipe gestures on content still work (the dock might not take much vertical space, but if finger is on dock and swiping it might not reach the ring content area; ensure user typically will swipe above the dock).
- [ ] **Styling & Consistency:** Use Tailwind for spacing and responsiveness. Possibly incorporate a shadcn/ui component if it simplifies (e.g., use a pre-built `Sheet` for mobile menu if needed, or `Button` styles for nav items for consistent focus outline, etc.). However, custom layout likely fine.
- [ ] **Logging:** Use the `onRingChange(oldId, newId)` hook to log whenever the user switches rings via the nav [21]. This can simply log to console or push to debug panel for now.
- [ ] **Test Nav:**
- Desktop: click each side panel button. Confirm the ring content changes appropriately and the new ring's nav item is highlighted. The transition effect (if implemented as fade) should be observed.
- Mobile: double-tap the dock, see the ring cycle to next. Test multiple sequential double-taps. Ensure it cycles through correctly back to start after the last ring.
- Also test mobile in a slower tapping scenario to ensure single taps don't inadvertently switch (if we implement only double-tap, single tap might do nothing which is okay).
- Verify that nav remains visible and doesn't interfere with scroll (nav clicks shouldn't scroll content, etc.).
- [ ] **Accessibility:** Label the nav appropriately (e.g., `nav` role or aria-label like "Section Navigation"). For side panel, each item should be focusable (maybe treat them as links or buttons) and have an accessible name (the section name). For dock, if it only cycles, maybe a single button with label "Next section: {name}". Ensure keyboard users can focus and activate these (we will allow keyboard ring switching via arrows too, but nav should be focusable as well).

Ring Switching & Transition Animations

Goal: Implement the logic for switching between rings when triggered (via nav or other means), including smooth **transition animations**. Initially, a simple fade transition will be used for polish, but

the system should be extensible to other types (cut, slide, background morph, composite) as outlined in the blueprint [32]. This feature handles unmounting one ring and introducing another with a visual effect, coordinating with background changes and ensuring state consistency during the swap.

Files/Modules to Create or Update:

- **Global State/Controller** – The `changeRing` function (as part of a context or utility module) will be implemented here. No separate file was defined for transitions, but this logic can reside in the RingsProvider or a new module like `src/rings/engine/TransitionManager.ts` if complexity grows.
- `BackgroundController.tsx` – may need updates to handle background transitions in sync (for transitions like `morphBackground` or to simply switch backgrounds when ring changes) [15].
- `RingView.tsx` – possibly adjustments to support being temporarily overlapped (for crossfade we might overlay two RingView components during transition).

Development Tasks:

- [ ] **Define RingTransition Options:** Create a TypeScript type or enum for transition types (`'cut' | 'fade' | 'slideAlongNewAxis' | 'morphBackground' | 'composite'`) with an optional duration, as specified in the blueprint [32]. This will be used as the parameter for `changeRing`.
- [ ] **Implement** `changeRing(nextId, transition)`**:**
- Locate the currently active ring and the target ring. If `nextId` is the same as current, do nothing.
- Determine transition type (default to `'fade'` if not provided for a nicer UX than a raw cut).
- For `'cut'`: Simply update state `activeRingId = nextId`. React will unmount the old RingView and mount the new one immediately. (This is a baseline functionality if animations are disabled or in reduced motion mode).
- For `'fade'`: Implement a crossfade:
  - Keep the old RingView mounted while mounting the new RingView on top of it (or underneath, but likely on top and initially hidden).
  - Perhaps use React state to render both: e.g., have an array of active rings, or a transient component that holds the outgoing ring.
  - Apply a CSS or Framer Motion animation: set old RingView opacity from 1 to 0, new one from 0 to 1 over `durationMs` (say 300–500ms).
  - Once complete, remove the old RingView from DOM.
  - This can be done with Framer Motion's `<AnimatePresence>` and variants, or manually with CSS classes and a timeout. Framer Motion might simplify it: define a variant for RingView on exit (opacity 0) and on enter (opacity 1).
- For `'slideAlongNewAxis'`: This is more complex (rotating the plane to the new axis while sliding). We may defer implementing this in detail. Possibly, one could animate the axisDeg rotation of the current ring to match the new ring before switching. This likely requires more physics – so mark as future enhancement.
- For `'morphBackground'`: Also advanced – would animate background only. Possibly as a future extension (like crossfade background or animate gradient).
- `'composite'`: combination of above (future).
- Ensure during any transition, user input (scrolling) on rings is disabled or ignored to prevent interference. Possibly set a flag `isTransitioning = true`.
- [ ] **Synchronize Background:** When switching rings, if backgrounds differ (e.g., color or image), include that in the transition:

- For fade: we can simply let each ring have its background, and as we crossfade the content, the background will effectively change under the fade. Alternatively, fade the background layer between old/new settings.
- If implementing `morphBackground` later, might do an animation on the background only.
- For now, ensure that after transition completes, `BackgroundController` knows the new active ring and updates accordingly (immediate or already done).
- [ ] **Update Active State & Clean Up:** Once the transition animation finishes, finalize the state:
- Set `activeRingId = nextId` if not already (some logic might set it at start or prefer end).
- Clear the transitioning flag.
- Unmount the old ring component if still mounted.
- Trigger the `onRingChange(oldId, newId)` hook after completion (or at start if we consider the moment of switch as trigger) [21].
- [ ] **Integration in Nav:** Ensure the SidePanel and Dock use `changeRing` with the appropriate transition. E.g., a desktop nav click might always call fade; mobile double-tap might just cut (since it's quick) or also fade.
- [ ] **Error Handling:** If `nextId` is not found in config, log an error (should not happen if nav is built from config).
- [ ] **Testing Transitions:**
- Try switching rings on desktop via nav: observe the old content fade out and new content fade in, with no flicker or intermediate blank frame. Items should appear in their correct positions.
- On mobile, double-tap the dock: ensure the transition works similarly (if using fade). Watch for performance on mobile – fade should be fine as it's just opacity.
- During transition, attempt a scroll (it should ideally do nothing until new ring is in place).
- Try rapid ring switches (like click two different nav items quickly): ensure state doesn't get stuck. Possibly disable nav interaction until one transition ends (you can enforce this via `isTransitioning` check).
- [ ] **Reduced Motion Consideration:** If user prefers reduced motion, skip animations:
- In `changeRing`, if a global `reducedMotion` flag is true, override any fancy transition to just do a `'cut'` (instant switch), or at most a very quick fade with no delay.
- This aligns with accessibility requirements (no prolonged animations if not desired).
- [ ] **Future Extensions:** Document in comments how one would implement slide or composite transitions in the future (e.g., using GSAP to animate the axis rotation and item positions before switching). This keeps the plan extensible.

Alternate Frame & Mini-Nav (AltFrame)

Goal: Provide an alternate full-screen frame to display content that doesn't fit within a ring item (for example, detailed project information or other pages). When triggered (likely by clicking a ring item), the AltFrame covers the ring UI and shows the content, with a mini navigation to return or switch context. This feature ensures the portfolio can show in-depth content while maintaining the ring navigation metaphor (via a "go-back" mini-nav, and possibly showing which ring it came from).

Files/Modules to Create or Update:

- `src/frames/AltFrame.tsx` – A component for the alternate content frame [34].
- (Reuse) `FrostedPanel.tsx` – Can be reused to create the mini-nav bar in the AltFrame.
- **Global State** – Use the `isAltFrameActive` boolean and possibly a reference to what content to show.

Development Tasks:

- [ ] **Implement AltFrame component:**
- Structure: It should cover the whole viewport (position fixed, z-index above ring and nav). Use a full-screen div or modal approach.
- Inside, include a content area (which could be just a placeholder or a route outlet if using a router for different pages in AltFrame).
- Include a **mini-nav header** at the top (or bottom) of AltFrame: a small bar containing a "back" button (to exit AltFrame) and perhaps a title or icon indicating the current section or content.
    - Use `FrostedPanel` for this mini-nav for consistency (a translucent bar).
    - The back button could be an icon (e.g., a left arrow from Lucide) or text "Back".
    - If the AltFrame is tied to a particular ring (say a project details from the "work" ring), it might show the ring name or item title.
- The AltFrame content can initially be static or a placeholder (like "Content goes here…") unless we know specific content to load. We can integrate with a routing scheme if one exists (e.g., using React Router to mount AltFrame for certain routes).
- [ ] **Triggering AltFrame:** Determine how AltFrame is activated:
- Most likely, clicking a **content** item in a ring (not a placeholder) should open its details in the AltFrame. For example, an item in the "Work" ring representing a project could open a detailed project view.
- Implement onClick in RingItem for `kind: 'content'` items that are designated to open AltFrame. This onClick would set global state `isAltFrameActive = true` and perhaps store a reference like `activeContentId` to know what to show.
- Alternatively, if using routes, onClick could navigate to a new route that triggers AltFrame.
- [ ] **Global State Handling:**
- In the global context or App state, add `isAltFrameActive` and possibly `altContent` info. When true, render the `<AltFrame />` component above the rest of the app (e.g., conditionally in App.tsx).
- Ensure when AltFrame is active, the underlying RingView and Nav might still exist but should be inert. We can add an overlay or simply rely on AltFrame covering it.
- Possibly dim or blur the background (optional enhancement: could apply a slight scale-down or blur to the ring behind AltFrame for effect).
- [ ] **Back Navigation:** When the user clicks the back button in AltFrame's mini-nav:
- Set `isAltFrameActive = false` to hide AltFrame.
- If we paused or altered anything in background, restore it (e.g., resume ring animations if they were paused).
- If focusing, return focus to the ring's item that launched the AltFrame (for accessibility, the blueprint suggests focus order should restore properly) [27].
- [ ] **Behavior and Style:** The AltFrame essentially acts like a separate page:
- No ring motion happens here (the blueprint explicitly notes "no ring motion inside" AltFrame [22]).
- We might still show the SidePanel if desired for navigation, or perhaps hide the main nav to reduce clutter. The blueprint implies AltFrame has its own mini-nav, so likely the main SidePanel/Dock might auto-hide when AltFrame is active (or at least be covered).
- Implementation: We can hide SidePanel/Dock via CSS or conditional render when AltFrame is on, to avoid user clicking another ring behind.
- [ ] **Animation:** Animate AltFrame appearance:
- Use Framer Motion to fade/slide it in. For example, slide from bottom or simply fade in the content and mini-nav.
- Similarly, animate out when closing (fade out).

- Keep animations short (~300ms) and respect reduced-motion (instantly show/hide if motion is reduced).
- [ ] **Testing AltFrame:**
- Simulate clicking a ring item (in a ring that we designate to open alt content). Verify that AltFrame appears, showing the expected content or placeholder.
- Click the back button, ensure AltFrame disappears and the ring UI is interactive again.
- Check that if AltFrame is open and the user tries to use nav (if we left it visible), either it doesn't work or is hidden. Likely better to hide main nav while in AltFrame to enforce focusing on content.
- Ensure that toggling AltFrame doesn't reset the ring state unexpectedly. The ring behind should remain where it was.
- [ ] **Extensibility:** In comments or documentation, note how AltFrame could be used for multiple different content pages (maybe via an `altFrameType` or loading different components based on what triggered it). For now, one AltFrame component can be used generically.
- [ ] **Accessibility:** The AltFrame mini-nav should be keyboard accessible (the back button focusable and with an aria-label "Go back"). When AltFrame opens, consider trapping focus within it (like a modal) so that Tab doesn't go to background elements. When it closes, restore focus to the last focused element in the main UI.

Background Effects & Controller

Goal: Create a system to manage dynamic backgrounds for each ring, enhancing visual appeal without interfering with content. Each ring can have a background specified (solid color, gradient, image, video, or even a React component for advanced effects). The BackgroundController will handle rendering the correct background for the active ring and updating it (for example, reacting to scroll progress if `followsIndex` is true) [35]. It ensures backgrounds transition smoothly on ring changes and remain decoupled from the ring movement (no physical scrolling unless explicitly coded, per spec).

Files/Modules to Create or Update:

- `src/rings/engine/BackgroundController.tsx` – A React component or utility that orchestrates backgrounds [15]. It may maintain a background element (like a full-screen div or canvas) and update its appearance based on active ring and ringIndexProgress.
- `rings.json` – Already contains background config for rings (mode, followsIndex, config), which this controller will consume [36].
- Possibly CSS or assets for specific backgrounds (e.g., a CSS class for a gradient palette or references to image/video files in assets).

Development Tasks:

- [ ] **Implement BackgroundController component:**
- Render a single full-viewport background element (e.g., a `<div id="background-layer">`) that is absolutely positioned (`inset-0`) behind all content (z-index below ring content).
- This component should be mounted once (e.g., in App) and listen to changes in `activeRingId` and ring progress.
- Based on the active ring's background spec:
  - If `mode: 'color'`: simply set the div's background color (use Tailwind class for known colors or inline style for dynamic).
  - If `mode: 'gradient'`: apply a CSS gradient. The `config` might specify a palette or angle. If palette is like "teal-red", define a gradient (you might have predefined classes or

use style like `background: linear-gradient(...)`). For dynamic effects, maybe adjust gradient stop based on progress (if followsIndex).
    - If `mode: 'image'`: set a background image (using an image URL from assets). Use `background-size: cover` etc. Possibly crossfade when changing images (to avoid flash).
    - If `mode: 'video'`: render a HTML5 <video> element behind, or a canvas. Autoplay it muted if needed, or handle playback based on ring progress (not trivial; possibly play/pause or scrub).
    - If `mode: 'shader' or 'react'`: These are advanced (shader might mean a WebGL canvas, react could mean a custom React component for background). Initially, we might not implement these, but keep the structure open. For now, we can ignore or treat them as no-op with a console note.
- Handle `followsIndex`: If true, the background should react to `ringIndexProgress`:
    - For color/gradient: we could interpolate colors as the user scrolls. For example, define two colors in config, one for start and one for end, and blend based on progress.
    - For image: maybe use two images and crossfade based on progress (like a slideshow effect). Or a parallax shift slightly.
    - For video: maybe seek the video according to progress (advanced).
    - Keep it simple initially: perhaps just log that "followsIndex" is true and plan an effect. Maybe move a gradient's angle or change opacity of an overlay to hint at motion.
- [ ] **State Management:** The BackgroundController needs to know:
- Current active ring's background settings (subscribe to activeRingId changes).
- The current ring's `ringIndexProgress` (which updates continuously as user scrolls).
- Implementation: possibly use the hooks we created. For example, RingEngine can call `onBackgroundTick(progress)` hook frequently (maybe throttle it) [21]. BackgroundController can listen to that (if using a context or event emitter). Or simpler: in the React tree, pass the ring's progress as a prop to BackgroundController:
    - If using context for ring state, perhaps provide a context value for current ring progress that BackgroundController consumes.
    - Alternatively, tie BackgroundController to the RingView – but since we want one background across rings, better to keep it at top level.
- When the ring switches, smoothly transition backgrounds:
    - If mode stays same (e.g., gradient to gradient), you can animate between the two configurations over some time. Possibly use a short CSS transition or Framer Motion to interpolate colors.
    - If drastically different (e.g., color to video), maybe do a crossfade: e.g., fade out the old background element while fading in a new one. One way: maintain two background elements and crossfade them on change (similar to ring fade).
    - For now, a simple approach: instantly switch on ring change (since ring transitions are already happening, a simultaneous fade might suffice).
- [ ] **Performance:** Make sure the background layer updates efficiently:
- Avoid heavy computations on every tick. If needed, throttle `onBackgroundTick` to, say, 30fps or when certain threshold changes in progress.
- For video backgrounds, ensure using low resolution or optimized codec if possible; pause the video when its ring is not active.
- Using CSS for gradients and colors is very cheap. If doing canvas or WebGL later, ensure it's not redrawing unnecessarily when not active.
- [ ] **Testing Backgrounds:**
- Set up distinct backgrounds in `rings.json` for two rings (e.g., one color, one gradient) for testing.
- Observe that when switching rings, the background changes accordingly (with or without fade).

- Scroll within a ring that has `followsIndex: true`, see if any reactive behavior is implemented. If we implement color interpolation, verify it changes subtly as you scroll.
- Ensure that background changes do not scroll with content (the spec: background does not physically slide with items) [37]. Our implementation should have the background fixed in place (which it is, since it's a separate layer not moving with ring items).
- [ ] **Logging:** Use the `onBackgroundTick(progress)` hook to log how progress changes. Possibly also log when background mode switches (old mode -> new mode) on ring change, to debug transitions.
- [ ] **AltFrame Background:** Decide if AltFrame uses the same background layer or its own styling. Possibly, AltFrame might simply overlay a solid background or blur the existing one. We can say AltFrame covers everything with its own backdrop (like a white background to make it like a separate page), unless we want the ring background to still be visible behind it faintly. This is a design choice; for now, we can have AltFrame just have a solid background or continue the same background if it looks nice.
- [ ] **Dark Mode/Theming:** If supporting dark mode, ensure color/gradient choices adapt or at least look acceptable. Use Tailwind CSS variables or conditional classes if needed to adjust backgrounds in dark mode (or define separate palettes in config for dark mode).

Accessibility & Input Fallbacks

Goal: Make the ring UI system accessible and usable for all users, including those using keyboard-only navigation or assistive technologies, and those who prefer reduced motion for comfort. This involves adding keyboard controls parallel to touch/mouse input, proper focus management, ARIA roles/labels for components, and honoring user preference for reduced motion by adjusting or disabling certain animations.

Files/Modules to Create or Update:

- **Keyboard Input Handling:** Could be integrated into the InputMapper or as global event handlers in the RingsProvider/App. No separate file, but modify existing input logic to include keyboard events.
- **Focus Management:** Possibly in RingView or Nav components – ensure the DOM structure allows logical tab order and update focus on relevant changes.
- **ARIA and Semantics:** Add attributes in JSX of nav, items, etc., no new files but important modifications.
- **Styles for reduced motion:** May adjust Tailwind classes or conditional logic in transitions (in changeRing, Snapper, etc.).

Development Tasks:

- [ ] **Keyboard Scrolling Controls:**
- Add an event listener for `keydown` (on window or a specific focusable wrapper). This can be done in the RingsProvider or a dedicated component that wraps the whole interactive area.
- Map keys:
  - `ArrowDown` or `PageDown`: advance the ring forward (like scroll down). This should call `RingEngine.updateIndex(delta)` with a fixed small delta (maybe one item's worth or a fraction). Essentially simulate a scroll gesture.
  - `ArrowUp` or `PageUp`: move ring backward similarly.
  - Consider also `ArrowRight/Left` if ring axis is horizontal? However, to keep it simple, down/up can be uniform controls regardless of axis (since axis might be diagonal, one

could argue left/right keys could map to horizontal movement if axisDeg ~0, but that complicates things. We'll stick to up/down and documentation can instruct).

- Ensure when these keys are pressed, they do not scroll the page (call `preventDefault` if needed, especially for PageDown).
- Only activate these when the ring or any of its items/nav is focused to avoid hijacking keys when user is, say, typing in a form.
- [ ] **Tab Focus Cycling:**
- The ring's items should be part of normal tab order. Likely, each RingItem can have `tabIndex={-1}` by default (so they're not individually focusable in normal tab flow), and we manage focus manually when needed.
- Alternatively, allow tabbing through items but that could be a lot if many items – perhaps better: when a ring is active, Tab could jump to the next item in visible order.
- A possible approach: handle Tab keydown manually:
    - If focus is on a ring item and Tab is pressed, move focus to the next item (maybe the next item in the ring sequence).
    - If on the last visible item, wrap to first visible (like how carousel accessibility can work).
    - This is complex; as a simpler measure, we might allow normal DOM flow: if items are rendered in order in the DOM, Tab will go through them anyway (even if off-screen). That might be okay given the ring likely has limited items.
    - We will ensure that after an idle snap, the centered item is the one that gets focus (this aligns with expected behavior).
- After a ring switch (via nav), explicitly focus the first item of the new ring (or a meaningful default item) [27] . We can do this in `changeRing` by using a ref or by setting state that RingView uses to autofocus on mount.
- [ ] **Screen Reader Considerations:**
- Mark up the ring structure in a meaningful way. Possibly use ARIA roles:
    - The ring could be a kind of carousel. We might give the RingView a role="region" with aria-label describing its content, or role="listbox" with each item as an option (though it's not exactly a typical listbox).
    - We can announce when the ring changes (e.g., using `aria-live` region to say "Switched to Work section").
    - Each nav button in SidePanel could be an actual `<button>` with an accessible name (the section name). That covers navigation for screen readers.
    - The Dock's double-tap is tricky for screen readers since double-tap is a touch gesture. For accessibility, we should provide an alternative: perhaps tapping it (single tap) for screen readers could announce how to cycle, or simply treat it as a "Next section" button (one tap activates it once).
- Ensure images have alt text if used (for content ring items).
- AltFrame content should be accessible as a normal page (heading structure, etc., depending on content).
- [ ] **prefers-reduced-motion:**
- Detect this via CSS (`@media (prefers-reduced-motion)`) or JS (`window.matchMedia`). We can set a boolean in global state like `reducedMotion` .
- Throughout the system, adjust behavior:
    - Replace the fade transitions with instant cut or minimal transition.
    - Snap immediately rather than animated.
    - Possibly disable any parallax or background follow effects. For example, if backgrounds were reacting to scroll, we might stop that (or greatly simplify it) when reducedMotion is true.
    - If using Tailwind, ensure any utility classes for animation (like if we had an animate-spin or something on items) are conditionally applied.

- Provide a manual toggle for reduced motion in settings (optional, but some users have OS setting which we read automatically).
- [ ] **Testing Keyboard & A11y:**
- Try using the interface with only keyboard:
    - Press Tab from page start: SidePanel's first nav item should be focusable (if we make them <button>s). Arrow down/up on nav should ideally not scroll page (the ring handles arrow keys when focus is in ring content area).
    - Press Enter/Space on a nav item: ring switches.
    - After switch, press Tab: focus should move into the ring's content (the first item).
    - Use ArrowDown/Up: ring should scroll as expected.
    - Press Tab through ring items: ensure it cycles and that after the last visible item it maybe goes to the next focusable (which could be the nav or browser UI).
    - Open AltFrame via keyboard: focus the item that triggers it and press Enter. AltFrame opens – its close button should receive focus or be next in tab order.
    - Close AltFrame and ensure focus goes back appropriately.
- Use a screen reader (if possible) to check labeling: nav announcements, item descriptions (maybe announce index in ring? Could be overkill).
- [ ] **Documentation:** Document in a README or Guidelines that these controls exist (for user help).
- [ ] **Clean Up & ARIA:** Make sure no interactive element is unreachable by keyboard and no crucial info is only conveyed visually. For instance, if we use color backgrounds to indicate section, ensure there's text or label for it.

Performance Optimization

Goal: Maintain smooth performance and frame rates by adhering to best practices for animations and rendering. As the ring system involves potentially heavy visuals (blur effects, many DOM elements, animations), we need to optimize transforms, avoid unnecessary reflows, and keep the experience fluid even on moderate devices (targeting an i5 CPU with no dedicated GPU, per the performance budget [38]). This section outlines performance-related tasks that cut across the implementation.

Files/Areas Affected: (No single file; these are cross-cutting concerns across RingView, BackgroundController, etc.)

Development Tasks:

- [ ] **Use GPU-accelerated transforms:** Ensure that moving items in the ring is done via CSS `transform: translate(...)/rotate(...)` rather than manipulating layout properties (like left/top in a way that forces reflow). The approach in RingView will use transforms, which keeps animations on the compositor thread [38].
- Also ensure any opacity changes (like fades) use CSS transitions or transform (which they do for opacity) to offload to GPU.
- [ ] **Minimize Layout Thrash:** Avoid queries of DOM layout (like `.offsetWidth` or `.scrollHeight`) inside frequently running loops (especially during scroll). If we need such values (e.g., container size), compute them once and cache if possible.
- In Snapper or Input handling, do not use any operation that forces layout (like reading element positions) every frame.
- [ ] **Efficient Backdrop Blur:** The frosted glass nav uses `backdrop-filter: blur`. This effect can be costly if over a large area or if many pixels. Our design constrains it to a panel, which is good [39]. Still:
- Ensure the panel is not larger than necessary (just the width of side panel or height of dock).

- Test on lower-end devices; if performance suffers, consider alternatives (like a semi-transparent solid background instead of heavy blur).
- [ ] **Limit Animation Work:** If we manually use `requestAnimationFrame` for any smooth motion (like Snapper or inertia), make sure to cancel it when not needed:
- If ring is idle and snapped, no RAF should be running until new input.
- If AltFrame is open, potentially pause ring RAF loops.
- Re-use a single RAF loop if possible for multiple animations to avoid stacking (e.g., one loop to update both ring position and background if needed).
- Consider using library utilities (Framer Motion and GSAP internally manage optimized loops).
- [ ] **Conditional Rendering / Virtualization:** If rings can have many items (say dozens), consider only rendering a subset at a time:
- We can decide to render, for example, only the items that are within +2 or +3 positions of the current index. Others could be removed or hidden. Since the ring loops, this is tricky but manageable by always keeping a window of items.
- As an initial step, if the number of items is small (the portfolio likely has maybe 5-10 items per ring), full rendering is fine. But keep this in mind and perhaps implement a simple check to not render placeholders far away from view.
- [ ] **Optimize Media Assets:**
- Use appropriately sized images for content (no giant images if displayed as small cards).
- Compress images (webp as blueprint suggests for placeholders).
- Defer loading of offscreen content: e.g., if an image is in a ring but currently 5 positions away, you might not load it until it nears view (using dynamic import or an `<img loading="lazy">` attribute may help).
- For video backgrounds, use short loops or compressed clips; ensure they don't play in background if not shown.
- [ ] **Memory Management:** Clean up event listeners on unmount (avoid memory leaks). Ensure that when rings unmount (on ring switch), any timers or RAF in RingEngine or Snapper are cancelled or re-attached to the new active ring's engine.
- If multiple RingEngines exist (one per ring), pause ones that are not active to save CPU (though they wouldn't be running if no input on them).
- [ ] **Testing Performance:**
- Use browser dev tools Performance monitor to watch FPS while interacting. Aim for 60fps on modern hardware.
- Profile memory to ensure it doesn't continuously grow (no leaks).
- Test on a mobile device or emulator to see if touch interactions remain smooth.
- Particularly watch the effect of blur – if it causes jank, consider enabling it only on higher-powered devices or simplifying its intensity.
- [ ] **Logging for Performance:** (Dev only) Use the debug panel or console to log any slow frames or choke points if detected. Possibly integrate a stats monitor (like a tiny FPS meter overlay) during development for tuning.
- [ ] **WebGL Consideration:** The blueprint hints at possibly using WebGL in future for backgrounds [38] . If that happens, ensure to only render WebGL when needed (like not every frame if static until input). For now, not applicable, but our architecture isolating background logic will help plug this in later.

Logging Hooks & Debug Panel

Goal: Enhance developer experience and maintainability by providing hooks for key events in the ring system and a debug interface to observe these events. This helps in tuning parameters (like snap timing, velocities) and verifying that interactions are working as expected. The debug panel will be a

dev-only UI overlay that logs events such as ring switches, index changes, snaps, etc., without cluttering the console excessively.

Files/Modules to Create or Update:

- **Event Hooks Definition:** (No separate file; define in context or engine) – e.g., in RingEngine or global context define the callbacks `onRingChange`, `onIndexChange`, `onIdleSnap`, `onBackgroundTick` as no-op by default or observable events [21].
- `src/components/dev/DebugPanel.tsx` – A component for the debug panel overlay (could be placed in a dev-only section of the app).
- Integration in the app – Only render DebugPanel in development mode (`import.meta.env.DEV` or similar flag).

Development Tasks:

- [ ] **Define Hook Signatures:** In the relevant places (likely the RingsProvider or RingEngine module), define function types for:
- `onRingChange?(oldRingId: string, newRingId: string): void` – called when active ring changes [21].
- `onIndexChange?(ringId: string, ringIndex: number, velocity: number): void` – called on any ring scroll update (could be high-frequency) [21].
- `onIdleSnap?(startIndex: number, targetIndex: number, mode: 'midpoint' | 'center'): void` – called when an idle snap occurs [21].
- `onBackgroundTick?(progress: number): void` – called when background progress updates (may be frequent as well) [21].
- These could be stored in a central object or simply called via context. Perhaps easier: the RingsProvider can hold these as callbacks that the developer can subscribe to (for now, we will wire them to the DebugPanel).
- [ ] **Implement Hook Triggers:** Insert calls in code:
- After `changeRing` completes successfully, call `onRingChange` with old and new IDs.
- In RingEngine.updateIndex (or wherever ringIndex is updated from input), call `onIndexChange` with current values. Throttle this if needed to avoid spamming (maybe only call if significant change or on animation frame).
- In Snapper when a snap animation is initiated or completed, call `onIdleSnap(startIdx, targetIdx, mode)`.
- In the background update loop, call `onBackgroundTick(progress)` perhaps at some interval (maybe tie it to requestAnimationFrame for smoothness).
- [ ] **Build DebugPanel UI:**
- Design it as a small overlay, possibly collapsible. For simplicity, it can be a fixed-position box in a corner (e.g., bottom-left) with a semi-transparent background and mono-spaced text.
- It should display a scrollable list of events logged. Each event could be a line of text like:
  - `[RingChange] work -> about`,
  - `[IndexChange] ring:about index:3.27 vel:120px/s`,
  - `[IdleSnap] from 2.7 to 2.5 (midpoint)`,
  - `[BackgroundTick] progress: 0.45`.
- Provide a toggle to show/hide details. Perhaps just a small button that says "Debug" to collapse/expand.
- Maybe provide checkboxes or filters to show only certain events (if too noisy, one might want to see only snaps and ring changes, not every index tick).

- Implement as a React component with internal state holding an array of log lines. It can subscribe to the hooks:
    - We might make the hooks actually update this component's state. E.g., pass functions from DebugPanel into the RingsProvider to assign as the hook callbacks.
- Add a "clear" button to empty the logs.
- Style with Tailwind for quick layout (e.g., text-xs for text size, max-h-60 overflow-y-scroll for the log area).
- [ ] **Dev-Only Rendering:** Ensure DebugPanel is only included in development builds:
- Use a condition like `{import.meta.env.DEV && <DebugPanel .../>}` in App.jsx.
- This prevents exposing internal logs to users in production.
- [ ] **Test DebugPanel:** During development, interact with the app:
- Scroll the ring: see multiple `IndexChange` events (could be many; if it's overwhelming, consider debouncing updates to maybe 10 times per second).
- Stop and let it snap: see an `IdleSnap` event.
- Switch rings via nav: see a `RingChange` event.
- Possibly see `BackgroundTick` events as well if implemented.
- Check the panel's usability: can you scroll through logs, clear them, hide the panel? Adjust as needed.
- [ ] **Logging Alternatives:** If a full panel is overkill, we could also integrate with Storybook or other tools as mentioned in baseline (not in-app). However, an in-app panel is convenient for dynamic interactions, so we proceed with it.
- [ ] **Maintainability:** Document in the code that these hooks are for dev purposes. If in the future more sophisticated state management is used (like Redux or Zustand), these might be replaced by devtools. But for now, this lightweight approach suffices.
- [ ] **Commit to Baseline Standards:** Ensure the DebugPanel code and hook usage do not violate lint rules (maybe exclude it from coverage if we had tests). Also, ensure all logs are removed or disabled in production build (which our conditional render covers).

Quality Assurance & Testing

Goal: Verify that the implemented Ring UI system meets all the expected behaviors and usability criteria. Below is a checklist of acceptance criteria (from the Rings UX Blueprint) and additional quality checks to ensure a polished result.

Acceptance Criteria Checklist:

- [ ] **A1 (Desktop Scroll):** On desktop, using the mouse **wheel** advances the ring along its configured axis smoothly, and pointer dragging is not required or enabled [14]. (Test: scrolling the wheel moves items; click-and-drag does nothing in this context.)
- [ ] **A2 (Mobile Swipe & Cycle):** On mobile, a **vertical swipe** gesture advances the ring appropriately, and a **dock double-tap** cycles to the next ring in a fixed order [14]. (Test on touch device or emulator: swipe moves ring; double-tap on dock changes rings.)
- [ ] **A3 (Endless Loop):** The ring of items is effectively **endless**, with no perceptible seam when wrapping around from the last item back to the first [40]. (Test: scroll continuously through multiple loops; observe no jump or blank gap.)
- [ ] **A4 (Idle Snap):** After the user stops interacting, when the idle timeout passes, the ring gently eases to the nearest **midpoint** alignment (or to a center of an item if it was almost centered) [41]. (Test: stop at various positions and confirm correct snap behavior after the delay.)
- [ ] **A5 (Background Behavior):** The background **reacts** to ring index progress if configured (e.g., a gradient shift) but does **not visually scroll** with the items [42]. (Test: observe background stays

static in position when ring moves; any reactive effect is subtle and decoupled from direct item motion.)

- [ ] **A6 (Nav Overlay):** The navigation panel (or dock) is always visible **above** the ring content, and ring items pass **behind** it with a frosted-glass effect [33]. (Test: item graphics are visible under the translucent nav; nav remains clickable and doesn't interfere with scrolling.)
- [ ] **A7 (AltFrame Functionality):** The AltFrame renders when invoked, showing its content with a mini-nav and a go-back control, and while it's active, no ring motion occurs in the background [43]. (Test: open alt frame, ensure ring stops responding; close it, ensure ring resumes.)
- [ ] **A8 (Accessibility):** Keyboard and reduced-motion modes are supported as specified [44]. Arrow keys or other designated keys can navigate the ring, focus is managed when switching rings or opening AltFrame, and enabling reduced motion results in significantly minimized animation and motion. (Test with OS setting for reduced motion; test full keyboard navigation loop.)
- [ ] **Code Quality:** All new code conforms to the project's linting and formatting rules (ESLint, Prettier). Components and modules have clear names, are placed in the correct folders per structure, and include JSDoc comments as appropriate (e.g., each component and complex function has a brief description).
- [ ] **Modularity:** Verify that engine logic is separated from UI (no direct DOM calls in RingEngine, etc.), and that components are reusable and not overly coupled. The structure should match the proposed folders (rings/engine, rings/components, nav, frames, etc.), making it easy for future developers to locate functionality.
- [ ] **Integration Testing:** Do a final run-through of the entire app flow:
- Load the app (desktop and mobile layouts).
- Scroll the landing ring, let it snap.
- Switch to each ring via nav, test content.
- Open and close AltFrame if applicable.
- Try a variety of input patterns (fast scroll, slow scroll, spamming ring switches, etc.) to check stability.
- [ ] **Performance Check:** Ensure that typical interactions do not cause noticeable lag or stutter. If any performance issue is observed, note it for optimization (e.g., if the debug panel is open it might slow down – test with it off as well).
- [ ] **Logging & Debug:** In development mode, confirm that the debug panel is capturing events correctly and can be used to diagnose any unexpected behavior.

By following this plan and completing all tasks, we will create a robust, polished Ring UI system that aligns with the design blueprint [45] and adheres to our professional development baseline [46] [47]. The result will be an interactive, maintainable portfolio feature showcasing modern UI animation techniques and solid architecture.

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] Rings_UI_blueprint.md
https://github.com/MiguelDiLalla/Portafolio_v02/blob/2e00227c88cd67dea99c5607a75bd54555f31fdb/docs/Rings_UI_blueprint.md

[46] [47] Professional_Baseline.md
https://github.com/MiguelDiLalla/Portafolio_v02/blob/2e00227c88cd67dea99c5607a75bd54555f31fdb/Professional_Baseline.md