ChatGPT

Donut of Attention – Current State & Transformation Report

Current Structure Review

HTML Organization: The interface is structured as a full-viewport canvas with overlaying UI layers. In the HTML (`index.html`), a `<div class="app">` wraps the HUD. A minimal **sidebar shell** (`<aside id="sidebarShell">`) is present as a universal container for membrane UI, though currently mostly empty except for a header ("Membrane") and an Overlay toggle [1]. The main interactive panels are defined in a **floating layer** container (`<div id="floatingLayer">`), each as a `<section>` with classes like `.panel` and `.membrane-panel`. Every panel has a unique `id` and a `data-membrane-name` (used for display and identification) [2] [3]. For example, panels like **Entry Door**, **Neurosity Crown**, **The Everything Chalice**, etc., are declared in HTML with their content and controls inside [4] [5]. The current design still includes a hidden **"ghost" sidebar** on the right (`<aside id="sidebarRight">`), meant for docking panels, but it's visually suppressed (no background or shadow) [6]. A hidden `<div id="floatingDockLayer">` at the bottom of the HTML is reserved for a future **bottom dock strip** (for "Desk mode") [7]. A special restore button (`#membraneRestoreButton`) floats on the left edge – presumably to recall collapsed membranes [8]. In summary, the HTML organizes panels in a central floating stack and uses side containers (sidebar shell, ghost sidebar, dock layer) for docking and future navigation modes.

CSS Layout & Styles: The CSS (e.g. `light.css`) ensures the **3D canvas** (#donutCanvas) covers the full viewport behind the UI [9], and it positions UI elements above it with proper z-index. Panels with the class `.membrane-panel` are given a translucent, draggable card style (using CSS variables for glass effects and shadows) – though in the latest iteration many panel backgrounds have been forced transparent for a more seamless look [10]. The sidebars are styled to be non-intrusive: the left sidebar shell and right ghost sidebar are fixed to the viewport edges but with `background: none` and no shadow [11] [12] to avoid covering the scene. Scrollbars in sidebars are hidden or minimal to keep a clean appearance [13]. Importantly, **resizing and responsive behavior** are considered – panels are resizable (`resize: both` enabled via CSS) and constrained by min/max dimensions for usability [14]. The CSS also includes a **menu-ring** system (radial menu) styles – e.g. `.menu-ring__slot` has pointer events enabled while the ring container is mostly passive [15] – hinting at a circular UI mode. Additionally, CSS rules exist for a **search interface** within the membrane UI: a `.membrane-search` class for a search field and an `.is-search-hidden` class to hide panels that don't match a filter [16]. Overall, the CSS sets up a flexible, dark-themed HUD with a focus on transparency and layering (consistent with the "holographic" aesthetic).

JavaScript Structure: The app's JS (`app.js`) uses a modular approach to initialize and control panels. All membrane panels are registered on startup using a helper (e.g. calling `initDockablePanel` for each panel element) [17]. This function wraps a panel with drag, dock, and pin controls. Indeed, each panel gets an attached **control pill** UI containing buttons for docking left/right, floating, and the new **Circle and Desk modes**, plus a close ("Off") button [18] [19]. For example, `initDockablePanel` creates a "pin slider" (Left –

Float – Right) control and adds a **Circle Mode toggle** button and a **Desk Mode** button ("bundle to desktop") into this pill [20] [21]. These controls let the user change a panel's state: pinning it to the left sidebar shell, undocking it to float, sending it to the right (ghost) sidebar, entering radial (circle) mode, or bundling it to the bottom strip. The JS manages docking by moving the panel's DOM node between containers: e.g. into the sidebar shell's body for left dock, into a hidden ghost sidebar for right dock, or into the floating layer for free float [22]. State is tracked in objects (like `panelDockState`) so positions and modes persist (often saved to `state.ui` which syncs to `localStorage`) [23] [24]. There is also a **Membrane Directory panel** (see HTML `#membraneDirectoryPanel`) that JS populates with a live list of all membranes and their statuses [25]. This directory, implemented as a regular membrane panel itself, includes a search bar and actions (via JS events) to filter membranes and perform quick docking or focusing [26] [27]. In short, the JavaScript ties the interface together by registering panels, handling user interactions (clicks, drags, keyboard shortcuts), and enforcing the logic for the modular "membrane" system. All major UI surfaces (panels like *Neurosity Crown*, *Solar Gate*, *Dynamics Lab*, etc.) are now modular and controlled through this system rather than a fixed sidebar tower [28].

UI System Elements: Key UI components include the **Sidebar Shell** (the left panel with "Membrane" header) which is intended as a universal sidebar for navigation and search. The **Membrane Directory** (accessible as a panel or via the shell) provides an overview of all panels and quick actions [25]. The **Floating Layer** is where undocked panels live – by default panels start as floating, draggable windows over the 3D donut. The **Ghost Sidebar** on the right acts as a hidden docking area – a remnant of the old design where panels could be pinned off-screen to hide them. It's currently collapsed via CSS/JS when empty [29]. The **Dock/Pin Controls** on each panel (hovered at the top or in the header) is an important part of the UI system: it allows switching a panel's location or mode in the UI with one click (for instance, pinning a panel to the shell will insert it into the sidebar's DOM, while floating releases it) [22]. Legacy modes like *"pin" and "dock"* have been extended: now the control pill includes a **radial toggle (Circle Mode)** and a **desktop bundle (Desk Mode)** in addition to left, right, or float positions [30] [21]. These new modes are part of the ongoing overhaul to introduce more dynamic, metaphorical layouts. Finally, keyboard shortcuts are beginning to appear in the system (e.g., **Ctrl/Cmd+L** to toggle Alignment Mode for an orthographic view [31]), indicating an intent to allow quick keyboard-driven UI toggles. All together, the current structure sets the stage for a **modular, holographic interface** where panels can be moved freely, grouped in new ways, and controlled via a unified sidebar/search and keyboard, rather than a rigid single sidebar of the past [32].

Misalignments and Required Corrections

Despite the progress toward modular panels, several parts of the system remain misaligned with the new design vision and need correction:

- **Ghost Sidebar Artifact:** The right "ghost" sidebar (an off-screen docking column) is a legacy element that doesn't cleanly fit the new membrane paradigm. It currently can appear as an empty dark band or gradient on the right edge when not in use [33] [29]. This ghost sidebar should be eliminated or repurposed – presently it's collapsed via CSS (`display: none !important` when `aria-hidden="true"`) and extra JS guards [34] [35], which are essentially hacks. Removing the ghost sidebar will simplify layout and avoid the "phantom" UI artifacts that were observed (a blank panel on the right) [29]. Any essential functionality (like parking a panel off-screen) should be handled through the new Desk mode or by a more intentional hiding mechanism, rather than a hidden sidebar.

- **"Sidebar Door" and Legacy Sidebar Behavior:** The earlier UI had a concept of a *sidebar door* – possibly a sliding reveal or mask on the sidebar. Traces of this remain (mentions of a sidebar scrim/door in notes) and were causing visual issues. The development journal notes that sidebar "door" artifacts were removed and the sidebar was made solid and always present [36] [37]. Any remaining code or CSS related to a sliding door animation or scrim overlay on the sidebar should be purged, as the new design calls for a persistent, minimal sidebar (no more peekaboo door behavior). The sidebar now should act as a stable membrane container, so legacy door logic is redundant.

- **Hardcoded Styles & Layout Hacks:** Some UI behaviors are implemented via hardcoded styles that need refactoring. For example, in the HTML the floating panel layer is initialized with `style="display: none;"` and then shown via script [38] – this can be fragile if scripts fail. Similarly, widths for sidebars are clamped in CSS (e.g., fixed max-width) which might not adapt well to very small or large screens [39]. These values should be reviewed to ensure responsiveness (perhaps use relative units or media queries for extreme sizes). The dev journal also documents multiple attempts at forcing transparent backgrounds by appending `!important` CSS rules to override unwanted styles [10] [40]. These indicate "fixes on top of fixes." A refactor should consolidate these overrides (remove redundant or contradictory rules) and ensure the base styles are correct (so we don't rely on `!important` everywhere). In short, clean up any inlined or hardcoded CSS/JS that was added as a quick patch, and implement a more systematic solution consistent with the new design (e.g., a single source of truth for panel styling, one authoritative setting to hide the ghost sidebar, etc.).

- **Brittle Event Bindings:** The current JS binds many UI controls manually and imperatively (e.g., adding individual `click` and `keydown` listeners for each pin button, each keyboard shortcut, etc.) [41] [42]. This results in a lot of verbose code and potential for oversight (e.g., one button might be missing a keydown handler for accessibility, or a global key listener might interfere with form inputs). A more declarative or unified approach is needed. For instance, the pin-slider could have been a set of radio inputs in HTML, which would handle focus and ARIA state more naturally, rather than manually updating `aria-checked` on each button press in JS. The current approach works but is "brittle" in that it requires careful handling of focus, key events, and state sync in many places. Similarly, keyboard shortcuts like the Alignment Mode toggle are added in multiple places (the journal shows duplicate bindings fixed by deduping) [43] [44]. We should centralize keyboard shortcut handling to avoid conflicts and ensure future shortcuts can be managed in one module. In summary, cleaning up event binding might involve using event delegation (one handler for a group of similar buttons), leveraging native form controls where possible, and consolidating global shortcuts.

- **Legacy Elements and Dead Code:** With the membrane system in place, some old elements are no longer used and should be removed to avoid confusion. For example, `#navBoardCustom` (an old navigation board) is explicitly hidden via CSS [45] – likely a vestige of a previous UI. The presence of `.menu-ring` classes and `handleMenuRingClick` in JS suggests an older radial menu implementation that might be superseded by the new circle mode logic. If those aren't part of the new plan, they should be pruned. The dev notes also mention that in `app.js` the legacy sidebar references (`sidebar` and `sidebarRight`) are set to `null` to disable old behaviors [38]. This is essentially dead code handling – those references and any code depending on them can be removed once we fully migrate to the new sidebar shell. Removing unused HTML elements (like any hidden menu scaffolds), old CSS (like redundant sidebar styling that's overridden anyway), and JS flags for

legacy behavior will streamline the codebase and prevent unintended interactions. Every element should have a clear purpose in the new modular design; anything else is technical debt to trim.

- **Missing or Incomplete Features:** A few parts of the intended design are still missing or only partially implemented. The **universal sidebar shell** is in the HTML but currently carries no content by default (it's even initialized with `is-hidden` class). Its promised behavior (search, filter, pin, focus) is only partially present – for instance, the search input is actually inside the Membrane Directory panel rather than in a global sidebar UI [26]. We need to decide whether the sidebar shell will directly host the Membrane Directory or some quick-access UI. If it's to be "universal," possibly it should contain the search and a list of panels at all times, instead of hiding that inside one panel. Likewise, **focus mode** for panels (bringing one panel to prominence) is not obviously implemented yet; we might need a mechanism (like a "focus" action in the directory or a double-click handler) to maximize a panel or dim others. The **Circle Mode** and **Desk Mode** buttons exist in the UI controls but their functionality is likely stubbed or minimal at this stage – these need full implementation (discussed below). Also, while dragging panels is possible, we might lack **touch gesture** support (for tablet users or future XR interactions) – an area to address in the rewrite. Finally, **keyboard navigation** through panels or quick panel switching (apart from a couple of shortcuts) is not fully realized – e.g., no "Cmd+K" quick-open exists yet. These missing pieces should be added to align with the project's goals of extensibility and accessibility.

- **Redundancy in UI Controls:** With the introduction of new UI paradigms (circle and desk), we should reevaluate if any older UI control is now redundant. For example, the "pin to left" vs "bundle to left sidebar" might overlap – if Circle Mode or Desk Mode provide alternate ways to group panels, perhaps pinning to the ghost sidebar is obsolete. The Membrane Directory panel includes an "Add Custom Membrane" button [46] for prototyping new panels – we should ensure this doesn't duplicate functionality available elsewhere (e.g., maybe the Solar Gate's "Create Membrane" does something similar [47]). Ensuring each control in the UI has a distinct, needed role will prevent confusion. Any duplicate ways to do the same thing should be streamlined in the new design.

In summary, the misalignments revolve around leftover ghost/door constructs [37], ad-hoc styling fixes, and incomplete integration of the new modular features. Cleaning these up and closing the feature gaps will ensure the system truly reflects the intended holographic, modular design (with nothing extraneous "leaking" in from the old model).

Modular Rewrite Roadmap

To achieve the target **fractal-holographic navigation system**, the refactor will progress in stages. Each stage focuses on a subsystem (shell, panel API, circle UI, desk UI, etc.) and we outline multiple possible implementation strategies ("suggestion cloud") for each. The goal is to use lightweight, modular approaches (favoring vanilla JS/CSS/HTML with minimal libraries) to maximize compatibility and flexibility. The roadmap also includes fallback ideas to maintain graceful degradation if advanced features aren't supported. Below is the staged plan:

Stage 1: Universal Sidebar Shell & Search

Goal: Establish a unified sidebar interface (likely using the existing `#sidebarShell`) that provides global navigation of membranes – including search, filter, pinning, and focus controls – in one consistent place. This will replace the old sidebar behavior and make membrane management accessible at all times.

- **Approach 1: Integrate Membrane Directory into Sidebar Shell (Vanilla Implementation).** In this strategy, we convert the sidebar shell into essentially a built-in Membrane Directory. Instead of treating the directory as just another panel, we can render the list of membranes directly inside `#sidebarShellBody` so it's always available. We could move the search input and list (`#membraneSearchInput` and the directory list) out of the panel section and into the sidebar shell's markup. The **toggle button** in the sidebar shell header ("Overlay" button) could be repurposed to show/hide this side panel entirely (if we want the ability to overlay or collapse the whole shell). The advantage here is simplicity – we already have the directory logic; we'd just initialize it on the shell instead of in a panel. The downside is we lose a bit of modularity (the directory as a panel goes away) and we'd need to ensure styling is adjusted since the directory would no longer be a floating panel but a permanent sidebar region. However, ARIA roles and semantics can be preserved (e.g., use `role="navigation"` on the shell) and the search input stays functional [26]. This approach keeps things mostly vanilla: no external library needed, just refactoring markup and some JS event targets (point them to the shell DOM instead of the panel).

- **Approach 2: Sidebar Shell as a** Toggleable Overlay **(Enhanced with Micro-Library for Search).** Another strategy is to treat the sidebar shell as an overlay that can slide in/out or pop over content when needed (for example, appear when the user presses **Cmd+K** or clicks a "Membranes" button). The shell would remain hidden until invoked, preserving screen real estate. For the search functionality inside it, we could enhance the basic substring filtering with a lightweight fuzzy search library (like Fuse.js or similar). This would allow the user to type partial or misspelled queries and still find the right panel. The search library (just a few KB gzipped) could index the `data-membrane-name` and maybe keywords for each panel, enabling a more powerful search than simple "contains" matching. Pros: better UX for finding panels (especially as the number grows), and the sidebar could animate in a polished way (CSS transitions or a small JS for slide-in). Cons: an added dependency and a bit more complexity – we'd have to load and maintain the search index. If we choose this, we ensure it's only activated on demand (so it doesn't slow initial load) and we provide a basic fallback if the lib fails (like default text filter).

- **Approach 3:** Web Component **for Membrane Directory.** For a highly modular solution, implement the sidebar directory as a custom element (e.g., `<membrane-directory>` web component). This component, when attached, could automatically render the search input and list of panels, and handle filtering internally. It could expose methods to refresh the list when panels are added/removed. The advantage is encapsulation: the HTML, CSS, and JS for the directory are bundled, making it easy to reuse or even move (for instance, if someday we wanted a second instance of the directory or a pop-out window). It could also shadow-DOM encapsulate styles so it doesn't clash. However, using web components adds complexity (browser support is good now, but we'd need polyfills for older browsers if needed) and debugging can be a bit harder. It might be overkill if we only use it in one place. A simpler variation is to use a template or class in our app.js to generate the directory UI, which is more traditional. The web component approach is only justified if we foresee the directory being reused or significantly self-contained.

- **Approach 4:** Minimalist Fallback.** Regardless of how fancy the sidebar gets, we should ensure basic functionality in no-JS scenarios or if the user's device is small. A fallback could be an always-visible list of panel links in plain HTML (e.g., a `<nav>` list of anchors or buttons that each correspond to a panel toggle). This list can be hidden by default and shown via a `<noscript>` block or simple CSS if JS is not running. It won't have dynamic filtering, but at least the user can scroll through and find a panel. This ensures that even in a degraded environment, the core "navigate to a membrane" is possible. For filtering without JS, we can't do much, but we could visually group the list or alphabetize it to help findability.

Focus & Pin Controls in the Shell: Alongside search, the sidebar shell should allow "pinning" and "focusing" panels easily. Here are a couple ideas:

- We could add **pin buttons or drag handles** in the directory list: e.g., each item in the Membrane Directory could have small icons to send that panel to left, right, desk, etc. (like a mini version of the control pill). This mirrors functionality that exists on the panel itself. Alternatively, a simpler focus mechanism: clicking an item in the list could automatically scroll the viewport to that panel or bring it forward (if it's floating). If the panel is hidden (off), clicking might toggle it on (make it visible in floating layer). Essentially, the directory would become a one-stop control center for all panels.

- Another idea is a **"focus mode"** toggle: perhaps a button on the sidebar shell to enter a mode where only one panel is visible (others minimize or fade). Implementation wise, this could add a class like `.is-focused` to one panel and `.is-dimmed` to all others, triggering CSS to shrink or fade out the non-focused ones. The user could then cycle focus via the directory or a keyboard shortcut. The advantage is a clear, distraction-free view when needed (useful in a "fractal" UI to drill into one layer). A fallback if CSS transforms are not reliable is to programmatically hide all other panels (store their states to restore later).

Pros & Cons: The integrated approach (1) keeps everything in one place and leverages existing code – it's straightforward but might blur the line between a panel and the shell. The overlay approach (2) gives a cleaner separation (shell opens on demand) and enhanced search, but introduces an extra lib (fuzzy search) and complexity in timing/UI animations. The web component idea (3) is forward-thinking and modular but possibly too heavy for our immediate needs. We should also consider hybrid approaches – e.g., initially go with Approach 1 for simplicity, and later enhance the search with a library if needed (since adding Fuse.js later is easy once structure is there). What's important is that the sidebar shell becomes the **knowledge hub** of the UI – it should always know which membranes exist and their status. Ensuring a responsive design (perhaps collapsing to icons or a hamburger menu on small screens) will be key for broad usability.

Stage 2: Panel API & Lifecycle

Goal: Refine how panels are defined, created, and managed in code – making it easier to add new panels (a clear API), and ensuring panels behave consistently (modularity in functionality). We want adding a new "membrane" to be as simple as providing its content and telling the system about it, rather than writing boilerplate for docking each time.

- **Approach 1: Data-Driven Panel Registration.** We can create a central **panels manifest** – e.g., an array of panel definitions, each with properties like `id`, `defaultMode` (float/dock), `defaultSlot`, `contentSrc` (if we load content via AJAX or have sub-component), etc. On

startup, instead of manually querying each `document.getElementById('somePanel')` and calling `initDockablePanel`, we iterate through this manifest. For each entry, if the element exists in the DOM, we initialize it; if not and it's a dynamic panel, we could even create it on the fly using a template. This approach treats panels as data which can be easily reconfigured or loaded conditionally. Pros: All panel info is in one place, easier to see all membranes at a glance or to toggle certain categories on/off (for example, disable a panel by removing from manifest). It also helps when passing control to AI or code generators – the manifest is a clear specification of what to build. Cons: We need to maintain the manifest; there's a bit of redundancy (we have HTML and a manifest entry for it). But we can mitigate that by using `data-*` attributes in HTML as the manifest (i.e., query all `.membrane-panel` elements and read their `data-` attributes to build the internal list). In fact, the current HTML already lists `data-membrane-name` and sometimes flags like `data-membrane-fixed` [4] – we can extend this with, say, `data-default-slot="left"` or similar. Then `initDockablePanel` can read those attributes instead of requiring separate arguments.

- **Approach 2: Panel Class/Module (Encapsulation).** Create a **Panel class** in JavaScript that encapsulates all behavior of a membrane panel. For example, `class MembranePanel { constructor(element) { ... } }` which attaches the control pill, sets up drag events, etc., for that element. Each panel element, upon initialization, would be associated with an instance of this class (which could be stored in a Map for reference). Methods like `panel.open()`, `panel.close()`, `panel.dock(position)` could be provided, making it easy to script panel behavior. This object-oriented approach can make the code more readable by grouping related functions (rather than a bunch of standalone functions operating on global state). It also allows us to easily apply certain behaviors to groups of panels (e.g., iterate over all `MembranePanel` instances to save their positions). Pros: Clear structure, easier to manage state per panel (each instance can hold its own last position, etc., instead of storing in parallel structures). Cons: Introduces a level of indirection (developers need to be comfortable with classes or factory functions). Also, we have to carefully handle the initial state that might come from `state.ui` (persisted data) – but that can be passed into the constructor or a `restore()` method. A lightweight alternative is to use factory functions (not ES6 classes) to similar effect, if we want to avoid `this` context.

- **Approach 3: Leverage Custom Elements for Panels.** Similar to the sidebar component idea, we could define a custom HTML element for panels, e.g., `<membrane-panel id="xyz" slot="floating">…</membrane-panel>`. The connectedCallback of this element's class could automatically call `initDockablePanel` logic on itself. This way, simply by using the `<membrane-panel>` tag, we get the behavior. We could even allow `<membrane-panel name="Chaos Monitor" default-slot="right">` and have those attributes map to internal state. The benefit is that adding a new panel is purely an HTML exercise – drop in the custom tag with the appropriate attributes and content, and the element's class does the rest (creating its control UI, hooking events). This strongly enforces modularity; each panel is a self-contained component on the DOM. The drawback is again complexity and maybe performance: dozens of custom element instances each doing setup might be harder to debug if something goes wrong, and if a developer is unfamiliar, the indirection could confuse. Also, not all panels are alike – some have special behaviors (e.g. the Entry Door panel might be "fixed" in place and not dockable), so the custom element would need options/conditions to handle those exceptions.

- **Approach 4: Simplify via Convention (Minimal JS).** If we want to avoid heavy refactoring, we can still improve things by establishing clear **conventions** for adding panels. For example: "To add a

panel, create a `<section class="panel membrane-panel" id="YourPanel">` in HTML with the appropriate data attributes. In app.js, add your panel's ID to the `registeredMembranePanels` array and call `initDockablePanel` on it (most default behaviors will Just Work). If your panel needs special logic (e.g., custom event handling), encapsulate that in a function `initYourPanel()` and call it after docking init." This is more of a documentation and cleanup task: we'd ensure every panel's initialization is gathered in one place in code (rather than scattered logic), perhaps grouping them by category. It might also involve removing any panel-specific hacks in the global code and moving them into that panel's init routine. While not as structurally fancy as classes or components, this convention-based approach is pragmatic. It relies on discipline: e.g., use consistent naming, always call the register function, etc., so that an AI tool like Codex can easily follow the pattern to create new panels or modify them. The pro here is minimal code changes – mostly reorganizing – and no added libraries. The con is that it's not enforcing structure as strongly as the other approaches; a lapse in convention can cause inconsistency.

Lifecycle Considerations: In all these approaches, consider panel **lifecycle**: creation, showing/hiding, destruction. We should include methods or patterns for **closing** a panel (e.g., the "Off" button uses `panel.remove()` or at least hides it). If panels can be dynamically created (via the Custom Membrane Builder), we need our system to handle that too (the current code does have `createCustomMembrane()` which likely uses `initDockablePanel` under the hood [48]). Ensuring that removing a panel cleans up its event listeners (to prevent memory leaks) is also important – a panel class can hold references to its events and remove them on destruction, for example.

Pros & Cons Recap: Data-driven registration (1) makes the system transparent and easy for external tools, but might duplicate info between HTML and JS. The Panel class approach (2) organizes code well and eases state management, at the cost of introducing OO structure. Custom elements (3) maximize encapsulation and HTML-driven dev, but add learning curve and possible overhead. Convention/documentation (4) is easiest to implement but relies on developer care rather than enforceable structure. A combined approach might serve best: for instance, use a manifest (data-driven) + a small Panel class for each to handle instance behavior. The manifest can feed into creating instances of Panel class. This way we separate *configuration* from *behavior*.

By solidifying the Panel API, we ensure **extensibility** – new creative panels (e.g. a "Quantum Overlay" or "AI Assistant" panel) can be plugged in without touching the core, just by declaring and registering them. This modularity is crucial for the project's growth and the eventual goal of handing off to an AI co-developer or a plugin system.

Stage 3: Circle Mode – Radial UI

Goal: Implement **Circle Mode**, a radial interface metaphor that aligns with the toroidal theme. In Circle Mode, UI options or panels would be arranged in a circle (or torus) shape, likely around the main donut visualization, providing a spatial, circular menu. This could be used for selecting membranes or switching contexts in a more spatial way than a list. It should feel "holographic," as if controls orbit the user's focus.

- **Approach 1: Radial Menu for Panel Shortcuts.** One straightforward way to introduce Circle Mode is as a **circular menu of important actions/panels**. For example, when Circle Mode is toggled (by clicking the circle button on a panel's control pill [30] or a global toggle), a ring of icons appears around the donut. Each icon could represent a membrane or a category (e.g., one for "Inputs", one

for “Analytics”, one for “Settings”). We can implement this by absolutely positioning elements in a circle. For instance, create a container `<div class="menu-ring">` that sits centered on the screen (perhaps overlaying the donut). Inside, place child elements for each slot around the ring. By using simple trigonometry, we assign each slot an angle and position: `left = cx + r*cos(angle), top = cy + r*sin(angle)`. This can be done in JS on toggle, or even pure CSS if we use `transform: rotate()` on a parent element systematically. We already have `.menu-ring__slot` and related classes in the CSS [49], which suggests an existing structure for such slots. We might revive or repurpose that. The benefit of doing it manually with JS/CSS is full control: we can animate the icons flying in, make the ring responsive (calculate radius based on viewport or number of items), and ensure hit areas are large for touch. The down-side is we need to handle many states (open/closed, active selection, etc.) and ensure it doesn’t conflict with the 3D canvas interaction (we should probably disable orbit controls while the menu is open, or have pointer events bypass to the menu).

- **Approach 2: Use SVG or Canvas Overlay.** Another approach is to draw the radial UI in an SVG or Canvas overlay for precision and possibly a more “3D” feel. For instance, an SVG circle with icons on its circumference could allow scalable vector graphics for each icon (maybe use `<image>` or `<text>` elements on the SVG). We could leverage a library like D3.js for positioning elements on an arc – but that’s heavy for this purpose. A lighter method: use `<svg>` with basic math for placement. The icons themselves could be interactive (SVG elements support click events). The advantage of an SVG approach is crisp scaling and easy rotation of the whole group if needed. We could even animate the circle rotating to bring a selected item to a front/top position (similar to a rotary dial). The cons: introducing SVG complexity and mixing DOM (SVG inside HTML) which can complicate styling. Also, accessibility might need extra attention (ARIA roles for menu/ menuitem may not directly apply to SVG elements without additional attributes).

- **Approach 3: 3D Ring of Panels (Experimental).** To truly embrace holography, we could attempt to place actual panel elements (or their icons) on a 3D ring around the donut itself. For example, using CSS3D transforms or Three.js CSS3DRenderer, each panel icon could be a plane in 3D space orbiting the donut. This is quite advanced: it would blur the line between the WebGL world and the HUD. Implementation could involve creating tiny thumbnail representations of each membrane (maybe just an icon or text) and using the same torus geometry to position them at intervals. When the user activates this mode (perhaps best in an immersive context), they could gaze at or click one of the orbiting items to select. The pro is a dazzling, futuristic UI that literally makes the UI part of the scene. The con is significant complexity and potential usability issues (small moving targets, readability of text at angles, etc.). This might be overkill for the core release, but it’s a creative direction to note for the “fractal-holographic” ideal.

- **Approach 4: Radial Layout for Sub-Components of a Panel.** Instead of (or in addition to) using Circle Mode as a global menu, we can apply the radial metaphor **within** certain panels. For instance, the *Field Command Deck* or *Everything Chalice* might present its options as a ring of toggles around a mini torus diagram. This approach means building a radial UI pattern that panel developers can reuse. It might involve a small library or custom function that, given a container and N items, will auto-arrange them in a circle. We can template this and encourage using it where it enhances UX. This is less about a single “mode” and more about a design pattern – nonetheless, implementing it now (for say the Everything Chalice presets or the color pickers) could give immediate holographic

feel. The advantage is improving existing UI with better spatial arrangement; the disadvantage is it doesn't address the "global navigation" aspect of Circle Mode.

Accessibility & Fallback: For any radial menu, we must ensure it's accessible via keyboard and screen reader. ARIA can define it as a menu with menuitems. We should allow arrow keys to navigate between items in the circle (e.g., left/right arrows rotate focus around the circle). A fallback if the radial display isn't available (e.g., on a text-only or very old browser) is to provide a linear menu list (maybe the same items but in a temporary `<ul>`). We can hide that list normally and only show it via `<noscript>` or if a CSS feature test fails for transforms.

Usage in practice: With Circle Mode active, the user might press a dedicated key (maybe **C** or click the circle icon) to open the radial menu. They then either hover over an option or use arrows to highlight one, and click/press Enter to confirm. That could, for example, open the "Everything Chalice" panel – implementing the scenario from the question: *"opening Everything Chalice from the radial ring."* In this case, one of the radial icons would represent the Everything Chalice membrane (perhaps using a chalice icon or the text "Everything"). Selecting it triggers the same function as if the user opened it from the directory or pressed its hotkey – the panel becomes visible/focused. After selection, the radial menu could auto-close (or remain if we anticipate multiple actions).

Pros & Cons: The explicit radial menu (Approach 1) is relatively easy and matches existing partial code (menu-ring classes), so it's a good starting point. It gives a new interaction without huge complexity. The SVG approach could yield prettier results but might be unnecessary if CSS can handle it. The 3D approach is a "wow factor" but not needed for core functionality (possibly a future enhancement when we integrate more XR). Weighing it, we likely implement a basic CSS/JS radial menu (Approach 1) first. It's touch-friendly (large icons around thumb reach if on tablet), and visually in line with the donut (a circle around a circle). We keep the design modular: the radial menu is invoked on demand and doesn't permanently alter layout, so the normal UI is unaffected when it's off.

Stage 4: Desk Mode – Bottom Strip UI

Goal: Implement **Desk Mode**, where panels can be bundled into a **bottom strip** (like a taskbar or "desk") for quick access when minimized. This provides a way to declutter the screen by collapsing membranes into icons or small bars at the screen bottom (metaphor: putting things down on your desk) and then restoring them as needed. It should be graceful (animated if possible) and intuitive (like how a dock or taskbar behaves in an OS).

- **Approach 1: Bottom Dock Container with Iconified Panels.** We already have a placeholder `<div id="floatingDockLayer" aria-live="polite"></div>` in the HTML [7]. We can transform this into the bottom dock. The idea: when a user clicks the "Desk" button (bundle) on a panel, we **move that panel's element** into the `floatingDockLayer` container and give it a special class (e.g., `.membrane-panel--desktop`) to apply dock-specific styles [50]. The CSS for docked panels can reduce their size dramatically – perhaps show only the panel's title or an icon. For example, the `.membrane-panel--desktop` rule might set a fixed small height/width and hide the panel's body overflow (this is already partially in CSS: `overflow: hidden` [50]). We could also append an icon or use the panel's existing icon (if any) or first letter as a representation. Essentially, each docked panel becomes an **icon tile** in the bottom container. By making `floatingDockLayer` a flex container with horizontal flow, these icons will line up in a strip. The user can click an icon (the panel in mini

form) to restore it. Restoration would involve moving it back to the floating layer (or appropriate dock if it was pinned before) and removing the `.membrane-panel--desktop` class so it regains full size. Pros: This approach uses actual panel elements as the icon, meaning state (like content) stays in the element even while docked. That's convenient (no need to recreate content on restore). We just need to be careful to **pause any heavy activity** in a panel while it's docked (for performance) – e.g., if a panel has a canvas or animation, perhaps suspend it when minimized. This approach is intuitive and largely DOM-manipulation based. Cons: If many panels are docked, the bottom strip might overflow; we'll need to allow scrolling the dock (horizontal scroll) or wrap to multiple lines (less ideal). We should also implement tooltips on hover or labels on focus for icons if only an icon is visible, so users know what each minimized panel is.

- **Approach 2: Unified "Desk" Panel.** Alternatively, when bundling, instead of keeping each panel separate, we could collect them into a single **Desk interface**. For example, clicking "Desk mode" on multiple panels adds shortcuts into a *Desk panel* that lives at the bottom. This Desk panel could show a row of buttons for each minimized item (with panel name or icon). Clicking a button in the Desk panel would re-open that item (and remove it from the desk list). Essentially this is like a start menu/taskbar hybrid. Implementation could be: a hidden "Desk" membrane (a panel) that, once something is bundled, automatically appears or highlights. We might reuse the Membrane Directory panel concept here but filtered to "currently minimized" items. Pros: This centralizes the bottom UI (one panel to manage rather than many tiny ones) and could present more info (like notifications or status of each minimized panel). Cons: It's more complex to implement – we'd need to sync state between panels and this Desk panel. It might also be a bit indirect (two clicks to restore: one to open desk, one to choose panel) unless the desk panel is always open. Also, having the Desk itself be a membrane panel might conflict with itself being minimized or managed.

- **Approach 3: Window Manager Analogy (Advanced).** We can think of Desk mode in terms of a window manager: minimize, maximize, etc. An advanced approach could leverage an existing micro-library or pattern for window management. For example, some libraries allow "minimizable windows" out of the box. However, importing a whole window manager might be overkill and not custom to our donut metaphor. Another idea: use CSS transitions to visually **shrink a panel into an icon** (like an animation) – e.g., animate the panel's width/height to a small square and anchor it at the dock position. This gives a clear hint where it went. This could be done with vanilla CSS if we know the final position. We could even implement a **drag to dock** interaction: user drags a panel toward the bottom; when it nears the dock, it snaps/minimizes. That would need careful pointer event coding but would feel natural (especially on touch: fling a panel down to put it away). For now, this is an optional enhancement to consider once the basics work.

- **Approach 4: Fallback – Simple Hide/Show.** If fancy animations or containers are problematic, at minimum, the Desk mode can behave like a simple hide/show toggle with a persistent button. For instance, clicking Desk could just hide the panel (display: none) and create a small `<button>` somewhere at bottom (maybe overlayed) with the panel's name. Clicking that button unhides the panel and removes itself. This is essentially a manual implementation of minimize/restore without a real dock container. It's not as scalable (lots of buttons could clutter) but is a straightforward fallback logic if the structured dock is causing issues.

Key Considerations: We want the desk strip to be **touch-friendly** – big enough icons or labels to tap easily. Also, multiple desked panels should be distinguishable. We can use the panel's `data-membrane-name` as

the label on the icon (maybe abbreviated if too long). The design should account for cases like 10+ items; possibly allow the strip to scroll or arrow through hidden ones. ARIA: treat the dock as a toolbar with buttons (each button being the minimized panel, labeled with the panel name for screen readers).

Another consideration is **persistence**: do we restore docked panels on page refresh? It might make sense to save the fact that "Panel X was docked" in state, so that on reload, it appears in the dock rather than popping open. This can be easily stored alongside position (e.g., `panelDockState[panelId].mode = 'desktop'` meaning it's in the desktop strip).

Pros & Cons: The direct container approach (1) is likely simplest with the current code, given the `floatingDockLayer` placeholder. It leverages existing panel elements, so minimal duplication. The unified Desk panel (2) is conceptually neat but might introduce unnecessary indirection. I lean towards Approach 1 as the primary plan: treat the dock as just another location a panel can reside (similar to left, right, float – now "desk"). In fact, our docking controller can be extended to a four-way switch: left, float, right, desk. We saw in code a hint: the pin slider currently has left, center (float), right, and separately there's a Desk button [51] [21]. Perhaps we unify that: the slider could even gain a fourth position or the Desk button triggers a similar state change.

Once implemented, the Desk mode will let users **collapse a field panel to the desk** with one click, for example. Visually, the Field panel might shrink into a small tile labeled "Field Deck" at the bottom. The user can continue working and later tap that tile to expand it back out, resuming where they left off.

Stage 5: Collapse/Restore Mechanism

Goal: Provide a robust mechanism to **collapse (hide/minimize)** membranes and **restore** them. This overlaps with Desk Mode but also covers panels being closed entirely and brought back (via directory or a restore button like the on-screen `#membraneRestoreButton`). Essentially, define what happens when a user clicks the "Off" button on a panel (or otherwise dismisses it) and how they can bring that panel back into view when needed.

- **Approach 1: Soft-Close with Restore Dots.** We could follow the pattern hinted by `#membraneRestoreButton` (the little dot button on the left edge) [52]. For instance, when a panel is "turned off," we don't remove it from the DOM or directory; we simply hide it (CSS `display: none` or an `.is-hidden` class). Meanwhile, we could spawn a small floating icon/dot at the edge of the screen (left for left-docked panels, right for right, bottom for desk, etc.) as a visual indicator of a hidden panel. The user can click that dot to restore the panel to its last known position. This approach was partially implemented as a swipe-return button for membranes. We can generalize it: whenever a panel is closed, create a restore widget. For consistency, we might incorporate this into the Membrane Directory instead (e.g., closed panels appear grayed out in the list and can be clicked to re-open). But a direct on-screen restore affordance is user-friendly (like minimizing to a icon as in Desk approach, or a sidebar tab). Pros: Quick one-click restore and clear indication of hidden content. Cons: If many are closed, having multiple restore buttons/dots could clutter the edges (one solution: have one restore button that cycles through or opens a menu of closed items).

- **Approach 2: Membrane Directory as the Sole Restoration UI.** In this method, closing a panel simply hides it, and the official way to bring it back is via the Membrane Directory (or sidebar shell search). The directory already knows about all panels, so it can list even the hidden ones (possibly

with a different icon or color). The user would scroll or search for the panel name and click an action (maybe the dock icons or just selecting it) to re-open. The benefit is simplicity – we reuse existing UI, no extra floating elements. It also encourages users to use the directory/search, reinforcing one central control hub. The downside is discoverability: a new user might not realize closing a panel means you must use the directory to get it back (as opposed to expecting a minimize icon somewhere). We can mitigate this with small UI hints (toast messages like "Panel X closed. Reopen from Membrane list or press Cmd+K to search.").

- **Approach 3: Window-Manager Style Minimize/Maximize.** If we take a window manager approach, "Off" might actually remove the panel element from the DOM (full close). We could then on restore either recreate it or have a hidden template to reinsert. This is more like how applications open/close windows fresh each time. It could help free memory if some panels are heavy (you unload them completely). But in our context, panels often represent ongoing settings or data (e.g., Chaos Monitor or Dynamics Lab), so likely we want them to persist hidden rather than be destroyed. That said, a middle ground: we could destroy certain types (maybe analytics panels that are fine to reset on reopen) but keep stateful ones around. This approach would need more housekeeping (saving state of panel inputs, etc. before removal), so it's probably unnecessary complexity unless memory proves to be an issue.

- **Approach 4: Collapsible Sections vs Full Panels.** Ensure we differentiate *collapsing a panel's internal section* from collapsing the panel itself. Some panels (like *Brain Waves* have a "Collapse" button for the internal graph section) [53] . Our system should not confuse those with the panel-level minimize. Panel-level collapse should be clearly defined (maybe the "Off" button icon is different from any internal collapse icons). For implementation, the panel-level collapse could simply call the same function as Desk bundling (but without moving it – if not using desk, just hide it). We should audit panels to avoid overlapping behaviors (the Brain Waves "Collapse" likely just toggles a CSS class to hide the canvas area inside the panel – that's local to that panel).

Restore triggers: We have a few triggers to plan: - Clicking a restore dot (Approach 1). - Selecting from Directory (Approach 2). - Possibly keyboard shortcuts (e.g., a shortcut to "restore last closed panel" akin to reopening a browser tab, or specific ones like Cmd+1,2,... to open specific panels if we assign quick slots – advanced idea).

We should also handle *when to remove restore indicators*. If a user closes a panel, then manually reopens it via directory, any floating restore button for it should disappear (to avoid confusion).

Animation and Feedback: A nice enhancement is to animate the panel fading out or sliding down when closed, and reversing that on open – signals state change. We can also provide a subtle sound or vibration (if applicable) on these actions for multimodal feedback (optional, but contributes to the user's sense of a tangible interface). At minimum, aria-live announcements (the HTML container has `aria-live="polite"` in some places like floatingDockLayer [7]) should announce "Membrane X closed" or "Membrane X restored" for screen readers.

Pros & Cons: The direct restore button approach is user-friendly but might need careful UI design to not clutter. The directory-only approach centralizes control but could frustrate some users who expect a one-click undo of close. Perhaps we can do both: e.g., show a temporary restore button for, say, 5 seconds after

closing (“Undo Close”), which fades out if not used, assuming the user will then go to directory anyway. That covers quick mistakes (oops, I closed it) and keeps UI clean otherwise.

Stage 6: Keyboard & Accessibility Enhancements

Goal: Expand keyboard controls and shortcuts for the interface, making it efficient for power users and accessible for those who can’t (or prefer not to) use a mouse. Also ensure overall accessibility (ARIA roles, focus management) is solid for the new interactions (search, radial menu, dock, etc.).

- **Approach 1: Define a Standard Shortcut Schema.** We should establish a set of keyboard shortcuts that cover the most common actions: e.g., **Cmd/Ctrl+K** to focus the search bar (bringing up the Membrane directory sidebar if hidden), **Esc** to close an open overlay (sidebar or radial menu), arrow keys for navigating radial or dock items, **Enter/Space** to activate a focused control, etc. In code, we can maintain a **central map** of shortcuts – an object like `{ 'Ctrl+K': openSearch, 'Ctrl+1': togglePanel1, ... }`. This map can be used with a tiny library like **Mousetrap** or we can write our own listener that parses `event.ctrlKey` / `event.key`. We’ve already done something similar for Ctrl+L (alignment) [54]. Extending it is straightforward. Pros: Having them in one place in code makes it easy to adjust and avoids duplicates. We also document these in the UI (for instance, tooltips for buttons can say “Overlay (Ctrl+K)” or we add a “Keyboard shortcuts” help screen). Cons: Potential conflicts with browser/system shortcuts (we must pick combos carefully to avoid overriding things like Ctrl+T or Ctrl+W). Also, on Mac vs Windows, conventions differ (Cmd vs Ctrl); our code should handle both where applicable.

- **Approach 2: Context-sensitive Keys.** Implementing shortcuts that depend on context can make the UI smoother. For example, when the radial menu is open, the arrow keys should navigate it (and maybe loop around). Or when focus is in the search box, pressing Down arrow could move focus to the results list (membranes list). We might need to add some JS to handle these when certain modals are active. For instance, if `sidebarShell` (search overlay) is open, hijack some keys: Enter could open the first result, etc. This can be done by adding event listeners at those times or by checking a global mode flag in the keydown handler. We should also ensure to *prevent default* where appropriate (e.g., Ctrl+K should `preventDefault()` so it doesn’t trigger the browser’s search if any).

- **Approach 3: Use ARIA for Focus Management.** A lot of keyboard accessibility can be achieved by proper ARIA roles and attributes. For instance, the Membrane Directory’s list of panels can be a `<ul role="listbox">` with each item as `<li role="option">`, which allows screen reader users to navigate with arrow keys automatically and hear each panel name announced. When the search input is focused and the user types, we can have it update an `aria-live` region summarizing the number of results or highlight the top result. The radial menu can be `role="menu"` with each icon `role="menuitem"`, and as we manage focus, ARIA will assist in announcing. We should ensure every interactive element is focusable (tabindex if not naturally focusable) and has an `aria-label` or text. The Desk icons, for example, might just be an icon visually, but we can give the button an `aria-label="Restore [Panel Name]"`. Also, when a panel itself is focused or activated, its heading could have `tabindex="-1"` and `aria-live` to announce that “Neurosity Crown panel focused” or similar, though that might be verbose. Perhaps

simpler: moving focus into the panel's first input or header when opened so that screen reader context naturally shifts.

- **Approach 4: Provide Alternate Input Methods (Future).** Beyond keyboard, think about other inputs aligned with the project's ethos: **BCI or gaze inputs**. The DEV_NOTES mentions fused BCI/gaze/dwell control for Solar Gate and intentions [55]. While implementing that fully is a larger task, we can lay groundwork by ensuring our UI elements have clearly defined events we can trigger programmatically. For example, if a brain-computer interface signals "open menu", we could call the same function as a keyboard shortcut would. Or a gaze dwell on a panel's edge could trigger it to dock. These are specialized, but by modularizing our control functions (so that UI actions are not only tied to clicks but callable from anywhere), we allow future integration. For now, a simpler alternate method: **touch gestures** – e.g., swipe from left to open sidebar, swipe down on a panel to desk it, long-press on a panel title to open its radial context menu. Each gesture can be implemented with Pointer Events API or a small gesture lib. The key is to keep these enhancements non-conflicting with basic operation (e.g., dragging a panel vs swiping it – perhaps a swipe could be distinguished by velocity or direction beyond a threshold).

Testing and Fallback: We should test that all critical operations can be done with keyboard alone. If any cannot, adjust design or provide a fallback. For example, if dragging is impossible with keyboard, we might allow arrow-key nudging of a focused panel (not typical, but could be helpful to precisely position a floating panel). At least provide a way to cycle through docking positions via keyboard – perhaps when a panel header has focus, pressing Left Arrow pins left, Right Arrow pins right, Down sends to desk, etc., as a hidden feature. This could use the same internal functions as clicking those buttons, just bound to key events when the panel has focus.

Pros & Cons: Investing in keyboard support (Approach 1 & 2) has big usability payoffs, with minimal downside except development time. ARIA roles (3) are essential for accessibility compliance – no real cons, just need careful implementation. The alternate inputs (4) are forward-looking; implementing them is mostly beneficial (no harm to existing input methods), but they can be postponed if time is short since they're enhancements. However, planning for them now (with modular design) saves effort later.

By the end of Stage 6, interacting with the Donut of Attention should be fluid whether using mouse, keyboard, or touch. A user should be able to hit a shortcut, type a query, hit enter, and jump to a membrane – or toggle modes without leaving the keyboard. This aligns with the intention of a creative tool that "gets out of your way" and lets you steer perception in a flow state.

Illustrative Usage Examples

Let's walk through a few example user interactions in the transformed system to illustrate how the new design empowers the user in a *holographic* and modular way:

- **Opening "The Everything Chalice" from a Radial Menu:** Imagine the user is in an immersive flow and wants to pull up the *Everything Chalice* panel (which provides a stacked cross-scale donut view). Instead of hunting through menus, they toggles **Circle Mode** by clicking the circular icon on the HUD (or pressing a hotkey). Immediately, a **radial menu** arcs around the central donut visualization, glowing slightly to indicate interactive nodes. Each node corresponds to a major membrane category or a specific panel. One of these nodes – perhaps under an "Analytics" category – is *The Everything*

Chalice (labelled or iconified accordingly). The user gazes at it (in XR) or moves the mouse to it; the node highlights under pointer. With a quick click (or a dwell selection via gaze), the Everything Chalice panel opens – the radial menu smoothly dissolves as the panel's controls fade into view. The panel appears floating, front-and-center. All of this felt like summoning an orb in a circle – a very *holographic* interaction where UI elements orbit the central focus and are selected in a spatial manner. If the user were instead using keyboard, they could press the Circle Mode key, arrow to cycle through radial options (hearing/seeing each name like "Everything Chalice" as focus moves), and press Enter to select [30]. This flow demonstrates the system's ability to present options in a **fractal circle** – fitting the metaphor of the torus (a circle of controls controlling circles of attention).

- **Collapsing a Field Panel to the Desk Strip:** The user has the *Background Grid & Field* panel open after tweaking various background overlays (Language of Light, Polar grid, etc.). It's a large panel taking up part of the screen, but now the user's done adjusting it for the moment. Rather than closing it completely (they'll likely need it later), they click the **Desk Mode** button on the panel's header controls [21]. Instantly, the panel shrinks with an animation – its borders compress and it "drops" to the bottom of the screen. On the bottom **desk strip**, a new small tile appears: a icon or small label saying "Background" (or an illustrative grid icon). The main view is now freed of that panel's clutter, showing more of the donut. The user continues to interact with other panels. Later, they decide to bring back the Background panel – they simply click the "Background" tile on the desk strip. The tile glows and expands back into the full panel, rising from the bottom to its former floating position (all the previous settings intact). This **bundle and unbundle** action is analogous to minimizing and restoring a window, but framed in our metaphoric language: the panel became a **"desk object"** (like a tool laid down on a desk), then was picked back up into the workspace. It's a smooth way to handle complex UI: at any given time the user can declutter by sweeping things onto the desk, knowing they're one click away from retrieval. Keyboard-wise, if the user pressed a shortcut (say, Ctrl+Down) to send a panel to the desk, the system could announce "Background panel minimized to desk" (via aria-live). They could then press a shortcut to focus the desk bar and arrow through icons to restore it, ensuring even without a mouse they have full control.

- **Using Cmd+K Search to Focus a Membrane:** The user wants to quickly adjust something in the *Neurosity Crown* panel (EEG settings), but that panel might be currently closed or somewhere off to the side. Instead of manually scanning the UI, they hit **Cmd+K** on their keyboard (or a dedicated "Search" button in the UI). Immediately, the **sidebar shell** slides out (or the search overlay pops up) with the search input focused [31]. They start typing "Crown". As they type, the Membrane Directory list underneath filters in real-time – other entries vanish and **Neurosity Crown** comes up as a top result (perhaps even auto-selected) [56]. The UI might highlight that result or allow the user to press Enter now. They do so, and the Crown panel springs into view (if it was closed, it opens; if it was docked elsewhere, it might flash to indicate it's that panel). The sidebar search overlay then auto-hides (optional, or maybe stays if we want multi-search). This experience is akin to the "command palette" in editors – a quick, textual way to summon any tool. It reinforces the **fractal-holographic** idea: no matter which layer or scale the desired control is in, a quick search surfaces it (holography: any piece of the system can be accessed from the whole via a small sample like a search term). Importantly, this interaction is efficient – it might take the user only a second or two – maintaining flow. For accessibility, the search results are navigable via Tab/Arrow, and each panel name is announced so a screen reader user can do the same. If no match is found, the system might politely say "No membranes match your query" (and perhaps offer to create one if that makes sense).

These scenarios show the transformed interface in action: **Circle Mode** offering an intuitive spatial navigator, **Desk Mode** providing modular rearrangement of the workspace, and **Universal Search/Sidebar** enabling instant context shifts. Each is designed to keep the user engaged with minimal friction, allowing them to fluidly steer their attention (true to the project name) through the various tools and views.

Optional Enhancements & Creative Ideas

As we refactor, there are opportunities to inject creative features that further align with the Donut of Attention's vision. These are not required for functionality, but could elevate the experience:

- **Fractal Sub-menus and Holographic Layers:** We can extend the radial menu concept to be recursive – e.g., selecting a category in the circle could spawn a second concentric circle of sub-options (a *fractal* menu). For instance, a first ring has "Inputs, Overlays, Metrics, Scenes" and if you select "Overlays", a second ring blooms outside it with "Grids, Nimbus, LoL, etc." This would truly embrace a holographic UI where menus have depth and layers. Implementation could leverage our radial positioning logic in multiple layers with distinct radii. We have to be cautious to keep it understandable, but visually it can reinforce the idea of drilling into sub-spaces.

- **Gestural Controls:** The new design could integrate gestures (especially if aiming for XR or touch). For example, a **two-finger twist** on the screen could trigger Circle Mode (imagine "dialing in" the circular menu), or a **swipe from bottom** could open the Desk strip (like pulling up a drawer). On desktop, mouse gestures (hold right-click and drag in a direction) could be bound to actions (not essential, but power-user friendly if documented). These gestures would complement the explicit buttons and might feel more natural in some contexts.

- **Dynamic Panel Grouping ("Membrane Orbits"):** Taking inspiration from the torus metaphor, panels could be allowed to form groups that **orbit together**. For example, if the user frequently uses three panels in conjunction (say Crown, Chaos Monitor, Dynamics Lab), they could pin them into a *group orbit*. This might manifest as those panels snapping together in the UI – possibly stacked or tabbed – and a single action could hide/show them all (as if they were one membrane with sub-pages). This is similar to grouping apps on a taskbar or tabs in a window. Achieving this might mean introducing a "group" container that can hold multiple panel sections. It's an advanced feature, but it could help manage many open panels by clustering them into thematic or workflow-based sets. The holographic analogy: each group is like a composite membrane that can be rotated into focus as a whole.

- **Visual Theming & Skins:** We could allow the user to switch the UI skin to different themes (light mode, high contrast, etc.). Given the creative nature of the project, even **thematic skins** like a "Solarized" theme or "80s neon grid" could be fun. This is mostly a CSS task (swap a CSS file or apply a theme class). Not critical for function, but important for accessibility (high contrast mode for visually impaired) and personalization (user feels more at home and in control of the instrument's look).

- **Persistence and Cloud Sync:** Ensure that all these UI configurations (docked left, desked, last search queries, etc.) are stored in state (which already persists to `localStorage` as `m-donut-state-v2` [57]). An enhancement would be to allow exporting/importing this state or even syncing via cloud if

this tool is used across devices. Then a user's holographic UI arrangement truly becomes portable – an advanced feature but in line with the idea of an "attention OS."

- **AI-assisted Arrangement:** If handing off to Codex or similar AI for layout, perhaps include an **auto-arrange** function where the system, given some criteria (most used panels, current task, etc.), suggests a layout (some panels to desk, some in circle). This could be a button "Smart Arrange" that users can choose if they feel overwhelmed by manual arrangement. The AI could use simple rules or learned preferences. This ties into creative exploration – the system could occasionally encourage a different perspective by rearranging the UI (with user permission).

- **Holographic Feedback:** Beyond the UI itself, use subtle effects to reinforce interactions – e.g., when a panel is focused or an important state changes, have a ripple effect on the torus or a glow on the donut corresponding to that action (linking UI and visualization). Example: opening a new membrane might briefly highlight the donut's corresponding layer or color (if applicable), creating a sense that the UI and content are one unified hologram.

These enhancements, while optional, can guide future development and keep the design language cohesive. They emphasize **modularity** (grouping and skins), **graceful fallback** (themes for accessibility), **touch-friendliness** (gestures), and **extensibility** (AI layout). They also maintain the central metaphors of holography and fractals – ensuring the UI not only functions well, but also inspires the user in the way it represents complex cognitive control in a playful, intuitive manner.

Key Divergences from the Old Model

The refactored design departs from the older Donut interface in several significant ways:

- **Single Sidebar → Modular Membranes:** The old monolithic sidebar ("tower") that listed all controls is gone. In its place is a **membrane system** where each control set is a floating or dockable panel [32]. This means no more scrolling one long sidebar; instead, users arrange panels freely. The new universal sidebar shell is only for navigation, not a fixed home for all controls.

- **No More Ghost/Hidden UI Elements:** The quirks like the ghost sidebar and sidebar "door" animations have been removed. The new design strives for **explicit UI** – if something is hidden, it's by user action (collapsed to desk or closed) and indicated clearly, rather than an artifact of a misaligned layout. The "ghost" right sidebar has effectively been replaced by intentional hiding/docking mechanisms.

- **Introduction of Circle & Desk Modes:** Two entirely new interaction modes are present. **Circle Mode (radial UI)** was not in the old interface – it's a fresh addition enabling circular menus and controls that align with the torus concept. **Desk Mode (bottom strip)** is another new concept; previously, panels could not be minimized to a bar – you either had them open or closed. Now we have a continuum of panel states (floating, docked, or iconified on the desk). This adds flexibility akin to an operating system UI, but within the web app context.

- **Unified Search/Command Palette:** In the old model, finding a control meant manually locating it in the sidebar or remembering a keyboard shortcut (which were few). The new model introduces a

Cmd+K style search palette, empowering users to jump to any part of the UI quickly. This significantly accelerates interaction for power users and mirrors modern productivity tools.

- **Panel Control Pill vs. Legacy Buttons:** Previously, controlling where a panel appears (pinning to side, etc.) may have been done via separate UI (or not at all). Now every panel has the **control pill** with a consistent set of icons for left, float, right, circle, desk, off [18]. This standardization is a divergence toward uniformity – no more special-case controls hidden in context menus; it's all on the panel header, symbol-driven, and identical across panels (with maybe minor exceptions for fixed ones).

- **Emphasis on Holographic Metaphor:** The new design leans heavily into the metaphors (fractal, holographic, toroidal). Features like radial menus and fractal menu layers were not present before and represent a conceptual shift to make the UI itself mirror the content's nature. The old UI, while themed appropriately, was still a conventional sidebar+panels setup. Now the UI's structure (circles, floating layers, nested groupings) *echoes the torus geometry and multi-scale attention model.* This is a philosophical divergence as much as a technical one.

- **Improved Responsiveness & Touch Support:** The refactored UI aims to be more responsive. The legacy design had fixed widths and was desktop-focused. We're moving to flexible layouts (using `clamp()` in CSS, etc.) [39] and adding touch-friendly hit targets and gestures. The result should be that the Donut of Attention can be used on tablets or small screens more comfortably, which was likely cumbersome before (the sidebar might have dominated on a small screen).

- **State Persistence and Custom Membranes:** The concept of *custom membranes* and persisting layouts is now front and center [58] [59]. In the old model, you got a fixed set of controls. Now users can create new membrane panels (via the Membrane Builder), which persist across sessions. The UI is no longer static; it's extensible at runtime. This is a fundamental change in how the system can grow – more like an operating system than a fixed app.

- **Removal of Clutter and Redundant Visuals:** Through the journal, we saw efforts to strip back unnecessary backgrounds, shadows, gradients that were part of the old aesthetic [60] [10]. The new UI has a cleaner look – transparent panels overlayed on the canvas with minimal chrome, letting the content shine. This modernizes the appearance and reduces cognitive load from the interface itself, matching the principle of "coherence over control" by not overwhelming the user with UI.

In summary, the new model shifts from a static, single-column UI to a **dynamic, multi-modal interface**. It diverges by giving users more control over the interface layout itself, integrating search and spatial navigation, and fully embracing the thematic metaphors. These changes not only improve usability but also reinforce the philosophy behind the Donut of Attention (making the tool itself feel like a living, fractal extension of the user's cognitive process rather than just a set of sliders).

Refactor Priorities Checklist

To implement the above transformation in a structured way, here is a **checklist of refactor tasks and priorities**:

- **Remove Ghost Sidebar**: Eliminate `#sidebarRight` from the layout or ensure it never renders unless needed. Clean up related CSS (no stray gradients) and JS (`sidebarRight` references set to null or deleted) [38]. Verify the canvas now spans the full width with no blank area [29].

- **Finalize Sidebar Shell Integration**: Populate the `#sidebarShell` with necessary elements – move search input and results list into it (or ensure Membrane Directory panel is accessible through it). Implement the sidebar toggle button to show/hide the shell overlay properly. Add "filter/pin/focus" controls in the shell if part of design (like filter checkboxes or focus mode toggle).

- **Consolidate Panel Initialization**: Refactor `app.js` panel init section to use a loop/manifest for registering membranes [17]. Ensure each panel's `data-` attributes drive its default docking, etc., so adding a new panel is one step (editing HTML or manifest) not many. Remove any hardcoded one-off init calls and replace with generalized approach.

- **Implement Circle Mode UI**: Develop the radial menu (HTML structure and CSS positioning). Connect the "Circle Mode" toggle button to open/close this menu [30]. Populate it with representative items (could be static for now or dynamically list key panels). Add keyboard navigation for it. Test that selecting an item triggers the expected panel open/focus.

- **Implement Desk Mode Behavior**: Activate the desk bundling. When "Desk" button clicked [21], move the panel to `#floatingDockLayer`, apply `.membrane-panel--desktop` style (ensuring it becomes a small icon/tile). Style the dock container as a bottom bar (fixed bottom, full width or centered). Make dock icons clickable to restore (move panel back). Handle multiple icons layout (flex-wrap or scroll). Test hiding/showing.

- **Off/Restore Mechanism**: Hook up the "Off" button on panels [61] to a close function. Decide on strategy (immediate hide vs. remove). Implement a restore path: e.g., generate a restore button at the side or mark the panel as closed in directory list. Possibly repurpose `#membraneRestoreButton` for this (make it appear when any panel closed, cycling through closed ones). Ensure that closed panels can be reopened via directory or restore click reliably.

- **Enhance Keyboard Shortcuts**: Map out new shortcuts (Cmd+K for search, Esc behavior, maybe Ctrl+M for circle mode, etc.). Add event listeners or use a library to support them. Guard against conflicts (don't trigger when typing in inputs). Provide visual hints (like underlining a letter on buttons or listing shortcuts in a help modal).

- **Focus Management & ARIA**: Review all interactive elements. Add `tabindex` to panel headers or control pills as needed so keyboard can reach them. Set appropriate `role` on radial menu (menu/menuitem) and dock bar (toolbar). Label icons with `aria-label`. Test with screen reader if possible (at least ensure the DOM order is logical – e.g., search input is followed by results list).

- **Clean Up CSS Overrides**: Since we are removing ghost sidebars and have new styling, purge the redundant CSS that was added to force transparency or hide scrims [40]. Consolidate panel styling in one place (e.g., define `.membrane-panel` background once as transparent glass). Remove old `.sidebar` styles not needed except maybe for the shell. Essentially, simplify the stylesheet now that we know what elements remain.

- **Testing on Different Devices**: Check the layout on a small screen (simulate mobile) – does the sidebar shell overlay properly or do we need it to full-screen on mobile? Are buttons large enough for touch? Maybe increase hit area for the tiny pin markers on touch devices. Also test on high-DPI and various browsers for quirks.

- **Membrane Builder & Custom Panels**: Verify that the **Membrane Builder** still works after changes. When a user creates a custom membrane, it should appear in the directory and be dockable/floatable like others [48]. Ensure our panel registration approach catches these dynamic panels too (maybe call initDockablePanel for the new element).

- **Persist and Load State**: Update persistence logic if needed – e.g., now include whether a panel is desked or circle mode active, etc., in `state.ui`. Test by moving some panels around, reloading page, and seeing if layout restores. Particularly, a panel left in desk or closed state should come back accordingly if we choose to persist that.

- **Documentation/Tooltips**: Update any in-app help (the Manual or tooltips) to reflect new UI. For example, change instructions that said "click the sidebar" to now say "open the Membrane menu (Cmd+K or left bar)" as appropriate. Possibly provide a quick on-boarding overlay highlighting Circle Mode and Desk Mode the first time, since those are brand new concepts.

- **Remove Deprecated Code**: After all above is working, do a pass to remove things we definitely don't use: old event handlers (e.g., any `handleMenuRingClick` if replaced), old CSS (any `#navBoardCustom` references), console logs or TODOs that are done, etc. This keeps the codebase lean for future contributors or AI agents.

Executing this checklist in order will transform the Donut of Attention interface into the intended state: a **modular, holographic control center**. Each step ensures that functionality is preserved or improved while aligning with the new design paradigm. Once complete, developers (human or AI) can much more easily extend the system – adding a new membrane or input method will be plug-and-play rather than surgery – which is exactly the outcome we want for sustainable innovation on this project.

---

[1] [2] [3] [4] [5] [6] [7] [8] [26] [27] [46] [52] [53] index.html
file://file_00000000d5e471f4bed14ce03845905a

[9] [11] [12] [13] [15] [16] [39] [45] [49] [50] light.css
file://file_0000000053b0724397e3e27f75f77315

[10] [29] [31] [33] [34] [35] [36] [37] [38] [40] [43] [44] [54] [60] dev_journal.md
file://file_0000000069c472438889b4f23862aae3

14 17 22 23 24 25 28 32 47 48 58 59 README.md
file://file_00000000f0fc7243b5aca9efd88d80f1

18 19 20 21 30 41 42 51 56 61 app.js
file://file_00000000c14c724395a89c9cc206c44c

55 57 DEV_NOTES.md
file://file_00000000c5cc7243b6a494b43e36244f