ChatGPT

Donudynomap UI Panel for Membrane

Overview

The **Donudynomap** panel is an interactive control interface for a donut-shaped gyroscope simulation in the Membrane app. It provides real-time manipulation of the **donut-gyro's** multiple spin axes and dynamic behaviors. Users can tweak rotation speeds (axial, poloidal, toroidal, orbital) and advanced coupling settings, with changes reflected instantly in the simulation. The panel starts compact with essential controls, and offers an expandable section for advanced flow dynamics. It also includes preset modes (both physical analogs and creative motion patterns) and a framework for saving custom presets. When no live gyro simulation is running, the panel falls back to a **demo mode** – showing a visual simulation of the donut's motion so users can explore the controls without an active gyro.

Primary Controls (Compact Layout)

In its default view, the Donudynomap panel presents a focused set of primary controls. These are the most commonly adjusted parameters for the donut-gyroscope's motion:

- **Axial Spin** – Adjusts rotation around the donut's central axis (like spinning a wheel). This slider/knob controls how fast the torus spins around its symmetry axis.
- **Poloidal Spin** – Controls rotation around the donut's minor circular axis (the small circle going through the donut's cross-section). This simulates the torus spinning through its "hole," akin to rotating around a ring's cross-sectional loop.
- **Toroidal Spin** – Controls rotation around the donut's major ring axis (the big loop through the center of the torus). This is like the donut turning around the vertical axis that runs through its hole.
- **Orbital Rotation** – Adjusts an external/orbital spin of the entire donut-gyro system. This can represent the donut precessing or orbiting around an external point or axis (for example, the torus gimbal turning around a base). In effect, the donut can be rotated in space around another axis, adding a fourth degree of freedom to its motion [1].

All four axes can act simultaneously, enabling complex gyroscopic motion. (For example, a classic spinning torus demo involves rotation around multiple axes – the torus's own circles plus additional rotations A and B about external axes [1].) Each of these controls would typically be a slider or dial with a neutral "zero" position (no rotation) and a range for positive/negative speeds. Changes to any slider update the simulation's parameters immediately, so the user sees the donut's motion respond in real time. Instant visual feedback is crucial – as the user adjusts a control, the linked rotation speed in the Three.js simulation updates on the fly (just as interactive GUI tools like Leva update a scene property live when a control is changed [2]).

- **Phase-Lock Ratio** – A specialized control to tune **coupled rotations**. It allows the user to lock or ratio-match certain spin axes. For instance, the UI might let the user set a ratio between two rotation speeds (e.g. axial vs. toroidal) so they maintain a synchronous relationship (like 2:1 or 3:2). This could be presented as a pair of linked dropdowns or a single slider that snaps to rational ratios. By adjusting this, users can enforce periodic alignment between spins – creating stable Lissajous-type motion patterns or resonances. For example, a 2:1 phase-lock might make the donut complete two axial spins for every one toroidal spin, yielding a repeating orientation pattern instead of drifting phases. There could also be a toggle to **enable/disable phase-lock**, so users can freely adjust speeds or have them constrained by the chosen ratio.

- **Coherence / Emission** – This control manages the **coherence** of the gyro's motion and any **emission effects** tied to it. "Coherence" could refer to how in-phase or stable the combined motions are – a high coherence setting might synchronize oscillations across axes, whereas low coherence introduces more independent or chaotic motion. "Emission" might tie into visual or audio output of the gyro (for example, light intensity or particle emission that increases with rotational alignment). In practice, this might be a dual-knob control or two sliders:
- *Coherence:* Could adjust a parameter in the simulation that blends chaotic vs. orderly rotation behavior (or variance in spin rate).
- *Emission:* Could control amplitude of a visual effect (glow, trail, sound) that the gyro emits, perhaps tied to how coherent the motion is. At low coherence, emission could be noisy or dim; at high coherence, it could stabilize or brighten.

These primary controls are arranged in a compact layout (e.g. a column of labeled sliders) so that a new user can immediately experiment with the gyro's fundamental behaviors. The design keeps it uncluttered initially, highlighting the key motion axes and basic tuning. Each control is labeled and possibly accompanied by a small icon or graphic (for example, an icon showing an arrow around the donut for each axis to illustrate the rotation axis). All changes take effect live by updating the Three.js simulation state, calling into the `createGyroSystem()` or related API to set the new parameters. The panel might also display the current value numerically next to each slider for precision.

Advanced Controls (Expandable Drawer)

For more experienced users or complex experiments, the Donudynomap panel includes an **Advanced** section hidden behind an expandable drawer (e.g. a chevron or "Advanced ▸" button). When expanded, this drawer reveals additional controls focusing on internal dynamics and coupling effects that go beyond simple spin rates:

- **Shear & Internal Flow** – Sliders to introduce or control shear forces and internal flow fields within the donut gyro. For instance, *Shear* might adjust a differential rotation between inner and outer parts of the torus (simulating a velocity gradient across the donut's cross-section). *Internal Flow* could control the speed or pattern of an internal fluid or energy circulation inside the donut. In a physical analogy, imagine the donut has an internal medium that can swirl as it spins – this slider could alter how that internal motion is coupled to the donut's solid body rotation. These parameters would influence the simulation's more nuanced behavior (perhaps affecting stability or the visual of how the donut's texture flows).
- **Axis Coupling Modes** – Options to couple the spin axes in different ways. This could be a dropdown or set of radio buttons in the advanced panel. For example: **Uncoupled** (each axis spins independently at its set rate), **Soft-Coupled** (axes influence each other slightly – e.g., increasing axial spin might gradually induce some toroidal motion, mimicking gyroscopic precession effects), or **Hard-Coupled** (strong gimbal lock or fixed ratio relationships beyond the simple phase-lock ratio control). These modes effectively let the user simulate different physical regimes – from an ideal

isolated gyro to one where rotations affect each other (like a gyroscope under torque where axial spin produces orbital precession).

- **Damping / Friction** – A slider to control overall damping. This can represent internal friction or air resistance slowing the spins. Turning up damping would cause the rotations to slow over time unless actively driven, allowing the user to see the gyro settle down. At zero damping, the gyro would spin indefinitely. This control is useful in advanced experimentation to see how the system behaves under energy loss conditions.
- **Field Coupling** – If the Membrane simulation supports magnetic or gravitational fields (for example, if the gyro is magnetically suspended or under gravity), the advanced panel could include controls for **field interactions**. E.g., a toggle or slider for *Gravity Influence* (to simulate a spinning top under gravity) or *Magnetic Lock* (if the donut has a magnetic axis that can lock to an external field). These are speculative, but the UI is structured to allow adding such controls as needed by the physics model.

All advanced controls are grouped in this collapsible section to keep the main UI clean. By default the advanced drawer is collapsed; the user can click "**Advanced**" to reveal these options. We ensure the drawer uses a scrollable panel or nested layout if many controls appear, to maintain a compact overall footprint. This design follows a common pattern of grouping secondary settings in collapsible panels (for example, libraries like Leva allow nesting controls in folders that can hide/show on toggle [3]).

Technically, the advanced controls might be implemented as a `<details>` HTML element or a custom React state (`showAdvanced` boolean that conditionally renders the advanced section). The styling would make it clear these are optional: possibly grayed subtitles or a slight indentation to differentiate from primary controls. Even when expanded, the advanced controls update the simulation in real time just like the primary ones – they tie into additional parameters of the `gyroSystem` (e.g. passing a `shearFactor` or setting a coupling mode flag in the simulation state).

Presets and Example Modes

To help users explore complex motion configurations quickly, the panel includes a set of **preset buttons**. These presets are organized by category, illustrating both real-world analogies and creative, metaphorical motions:

- **Physical Analogs:** Presets that mimic real gyroscopic systems or familiar physical objects. For example, **"Spinning Top"** – this preset configures the donut-gyro to behave like a classic spinning top toy. It might set a high axial spin (fast rotation around the vertical axis), a slight tilt, and enable an orbital precession at a slow rate (simulating how a top precesses under gravity). Other possible physical presets could be *"Orbital Gyro"* (where the donut orbits like a satellite with synced spin) or *"Engine Flywheel"* (very high toroidal spin, minimal wobble). These give users an intuitive starting point, as if selecting a known physical scenario. Clicking "Spinning Top" would instantly adjust all relevant controls (sliders jump to the preset values) and the simulation state updates accordingly to demonstrate that motion.
- **Metaphorical Modes:** Presets with creative names that illustrate a style of motion rather than a real object. For example, **"Breathing Precession"** – a mode where the donut's motion appears to "breathe." This could mean the axial spin speed oscillates rhythmically (like inhaling/exhaling), and the toroidal tilt gently rocks back and forth, creating a hypnotic precession that grows and recedes. In this preset, phase-lock might be tuned to a 1:1 ratio but with a slight phase offset that varies

sinusoidally, producing a pulsing motion. Another metaphorical preset might be *“Chaotic Wobble”* (introducing low coherence and a random wobble within a stable spin) or *“Laser Wheel”* (high coherence, with emission turned up to simulate a bright, steady disc). These presets inspire experimentation by showing off dynamic patterns with one click.

- **User-Defined Presets:** The UI is designed to accommodate custom presets saved by the user. Although full saving functionality might be slated for a future update, we lay the groundwork in the panel now. There could be a **“Save Preset”** button (perhaps appearing after the preset list or in a menu) that captures the current settings of all controls. Internally, this would serialize the values of all parameters (axial, poloidal, etc., plus advanced settings) into a preset object. In a future version, these could be named and stored (e.g. saved to local storage or the Membrane app’s state). For now, the button might open a prompt to name the preset and simply log it or store it in a temporary list. Likewise, a **“User Presets”** section (initially empty) can list any saved configurations for recall. The code structure anticipates this by providing functions to **save the current state** and **load a preset**. (This is conceptually similar to GUI tools like dat.GUI or lil-gui, which allow saving the current controller values and reloading them later [4].)

The presets UI might be presented as a row of buttons or a dropdown menu. A clean approach is to have category tabs or accordions: e.g. clicking “Physical Analogs” reveals buttons for Spinning Top (and others), and “Metaphorical” reveals Breathing Precession, etc. However, given only a few presets to start, a simpler list with headings can work:

```
Presets:
- Physical Analogs: [Spinning Top] [Gyro Orbit] ...
- Metaphorical: [Breathing Precession] [Chaotic Wobble] ...
- User-Defined: (none saved yet)  [Save Current as Preset]
```

Clicking a preset immediately applies its values to all controls (triggering the same update mechanisms as user adjustments). To provide a smooth user experience, applying a preset could animate the sliders to the new positions over a brief moment, so the change is visible. For the initial implementation, instantaneous jumps are fine. The preset system makes it easy to demonstrate the panel’s capabilities: for instance, a user can click through several presets to see dramatically different gyro behaviors without manual tweaking.

Demo Mode Behavior

When the gyro simulation is not active or no `gyroSystem` is connected, the Donudynomap panel enters a **Demo Mode**. In demo mode, the panel’s controls remain interactive but instead of controlling a real simulation, they drive a local simulated visualization. This ensures the UI is not empty or static when the actual gyro is off – the user can still explore the controls and see *some* response, which enhances learnability.

How Demo Mode Works:
- The panel detects if the `createGyroSystem()` simulation instance is absent or marked inactive (perhaps via a prop like `gyroActive = false`). If so, a **fallback Three.js scene** or animated graphic is used. For example, the panel could display a small canvas showing a 3D torus model. This torus isn’t the real simulation but a mock that responds to the panel controls. When the user moves the sliders, the torus in this mini-canvas animates accordingly (rotating about the selected axes with the set speeds). This gives

visual feedback of what *would* happen.
- The demo torus might have a pre-scripted loop if no interaction is happening – e.g. slowly tumbling by default. The moment the user adjusts a control, the demo takes those inputs and applies them so the user sees the effect. Essentially, the panel contains a lightweight gyro simulation for demonstration. If the real simulation becomes available (say the user powers on the actual gyro), the panel seamlessly switches to controlling the real `gyroSystem` and the demo preview can hide or sync to the real motion.
- There is an indicator in the UI when demo mode is active. For instance, an overlay label "Demo Mode (gyro offline)" could appear over the 3D preview, or the panel's background might change subtly. This reminds the user that they are not affecting the real system. The primary goal is to **showcase the controls** in action even if the backend is not running.

Implementing the demo mode might involve conditionally rendering a `<Canvas>` (from react-three-fiber) inside the panel component when `gyroSystem` is null/inactive. We could reuse the same Torus geometry and rotation logic but isolate it within the panel. For example, a React state `isDemo = !gyroSystem.active` can control this. In demo mode, an internal animation loop (using `requestAnimationFrame` or Three.js clock) would rotate the torus based on the current control state. The code ensures that when `gyroSystem` connects, the demo stops updating and all control changes route to the real simulation instead.

This approach provides a smooth user experience: the panel is always alive and illustrative. A user can learn what "poloidal vs. toroidal" means by seeing the effect on the demo torus immediately. It's also useful for presentations – one can show off the Donudynomap panel's capabilities without needing the full simulation running, using the demo as a stand-in.

Integration with Membrane Simulation

Integrating this panel into the **Membrane** app involves connecting the UI state to the `createGyroSystem()` simulation object and registering the panel in the app's UI system. The Membrane app likely has a context or store where the simulation lives. The Donudynomap component will fetch or be passed a reference to the gyro simulation instance so it can call its update methods.

Real-time Control Wiring: Each control in the panel triggers an update to the simulation as the user manipulates it. For continuous inputs like sliders, we use events or hooks to continuously stream changes. In React, this can be done by an `onChange` handler that updates local state (for the UI slider position) and also immediately calls a simulation update function. Alternatively, a small debounce can be used for performance if needed. The simulation might expose methods like `gyroSystem.setAxialSpeed(value)` or a generic `gyroSystem.update({ axial: value, poloidal: value2, ... })`. These would adjust the internal state that the Three.js animation loop uses. For example, if the simulation's update loop rotates the torus by `axialSpeed * deltaTime` each frame, changing that speed value will instantly alter the rotation rate visible on screen. This is analogous to how adjusting a parameter in a Three.js GUI can immediately change the scene – e.g., a GUI controlling background color will reflect on the canvas instantly [2].

To ensure smooth updates, the React component can utilize `useEffect` hooks. For instance, whenever our local state for a control changes, an effect calls the corresponding simulation update. If multiple controls change frequently, we might batch updates or have a single effect watching an object of all parameters. Here's a conceptual snippet illustrating this connection:

```
// Pseudo-code inside Donudynomap component
const [axial, setAxial] = useState(0);
const [poloidal, setPoloidal] = useState(0);
const [toroidal, setToroidal] = useState(0);
const [orbital, setOrbital] = useState(0);
// ... other states for phaseLock, coherence, etc.

useEffect(() => {
  if (!gyroSystem) return;
  // Update the live gyro system with new parameters:
  gyroSystem.updateSpinRates({
    axial, poloidal, toroidal, orbital,
    phaseLockRatio: phaseLockEnabled ? phaseLockValue : null,
    coherence: coherenceValue,
    emission: emissionValue
  });
}, [axial, poloidal, toroidal, orbital, phaseLockValue, coherenceValue,
emissionValue]);
```

In the above pseudo-code, `gyroSystem.updateSpinRates` is an imagined API call that applies the new values. The effect runs any time a relevant state changes, thereby keeping the simulation in sync with the UI. (If the simulation API expects individual calls, we could call separate setters in each onChange instead.)

Membrane Panel Registration: To embed this into Membrane's panel system, we would register the component so it appears in the UI. Depending on Membrane's architecture, this could mean adding it to a panel manager or a route. For example, if Membrane has a sidebar of panels, we might do:

```
import { registerPanel } from 'membrane-panels';  // hypothetical API
import DonudynomapPanel from './DonudynomapPanel';

registerPanel({
  id: 'donudynomap-controls',
  title: 'Donudynomap',
  component: DonudynomapPanel
});
```

This snippet conceptually shows adding our panel to Membrane's UI. In practice, the panel would be rendered within Membrane's existing layout (perhaps as a collapsible side panel or a tab within the simulation view). The panel should retrieve the `gyroSystem` instance – possibly via React context (`useContext(GyroContext)`) or via props if the parent passes it in.

Finally, ensure that the Three.js part of the simulation is properly initialized. The `createGyroSystem()` might be called when the simulation starts, returning an object with the torus Mesh and controlling functions. The Donudynomap panel could call `createGyroSystem()` itself if needed (for example, when entering demo mode or if the simulation isn't already running). But typically, Membrane might handle creation elsewhere and just provide the panel a reference. We'll integrate accordingly: e.g., the panel might do `const gyroSystem = useGyroSystem()` (using a custom hook provided by Membrane that gives access to the simulation).

Code Implementation Outline (React + Three.js)

Below is a structured outline of how the **DonudynomapPanel** component could be implemented with React and Three.js (using react-three-fiber for the 3D view). It's written as a combination of pseudo-code and real code to convey structure:

```
import React, { useState, useEffect, useContext } from 'react';
// Import Three.js or fiber components as needed
// import { Canvas, useFrame } from '@react-three/fiber';  // for demo mode
rendering
// import { Torus } from '@react-three/drei';              // maybe a premade
Torus component

// Context or prop to get the gyro simulation instance
import { GyroSystemContext } from '../membrane/simulation';

function DonudynomapPanel() {
  const gyroSystem = useContext(GyroSystemContext);  // get the simulation
(could be null if inactive)

  // State for primary controls
  const [axialSpin, setAxialSpin] = useState(0);
  const [poloidalSpin, setPoloidalSpin] = useState(0);
  const [toroidalSpin, setToroidalSpin] = useState(0);
  const [orbitalSpin, setOrbitalSpin] = useState(0);
  // State for advanced controls
  const [phaseLock, setPhaseLock] = useState({ enabled: false, ratio: 1 });
  const [coherence, setCoherence] = useState(1.0);
  const [emission, setEmission] = useState(0.5);
  const [shear, setShear] = useState(0.0);
  const [couplingMode, setCouplingMode] = useState('uncoupled');
  // UI state for showing advanced section
  const [showAdvanced, setShowAdvanced] = useState(false);
  // Manage demo mode state
  const isDemo = !gyroSystem || !gyroSystem.active;

  // Effect: whenever any control changes, update the real gyro system
  useEffect(() => {
    if (!gyroSystem) return;
    // Update basic spins
    gyroSystem.setSpinRates({
      axial: axialSpin,
      poloidal: poloidalSpin,
      toroidal: toroidalSpin,
      orbital: orbitalSpin
    });
    // Update phase lock and coherence/emission if applicable in simulation
    gyroSystem.setPhaseLock(phaseLock.enabled ? phaseLock.ratio : null);
    gyroSystem.setCoherence(coherence);
    gyroSystem.setEmission(emission);
    // Update advanced coupling
    gyroSystem.setShear(shear);
    gyroSystem.setCouplingMode(couplingMode);
  }, [axialSpin, poloidalSpin, toroidalSpin, orbitalSpin, phaseLock, coherence,
emission, shear, couplingMode, gyroSystem]);

  // Preset apply function
  const applyPreset = (preset) => {
    // preset would be an object like { axial:..., poloidal:..., toroidal:...,
orbital:..., phaseLock:..., coherence:..., shear:..., couplingMode:... }
    setAxialSpin(preset.axial);
    setPoloidalSpin(preset.poloidal);
    setToroidalSpin(preset.toroidal);
    setOrbitalSpin(preset.orbital);
    if (preset.phaseLockRatio !== undefined) {
      setPhaseLock({ enabled: true, ratio: preset.phaseLockRatio });
    }
    setCoherence(preset.coherence ?? 1.0);
    setEmission(preset.emission ?? 0.5);
    setShear(preset.shear ?? 0.0);
    setCouplingMode(preset.couplingMode ?? 'uncoupled');
    // The effect above will catch these state changes and propagate to
gyroSystem or demo.
  };

  // Example preset definitions
  const presets = {
    spinningTop: {
      axial: 5.0, poloidal: 0.0, toroidal: 0.1, orbital: 0.1,
      phaseLockRatio: null, coherence: 1.0, emission: 0.3,
      shear: 0.0, couplingMode: 'uncoupled'
    },
    breathingPrecession: {
      axial: 2.0, poloidal: 0.5, toroidal: 2.0, orbital: 0.0,
      phaseLockRatio: 1, coherence: 0.8, emission: 0.7,
      shear: 0.2, couplingMode: 'soft'
    }
    // ... other presets
  };

  // Render UI elements
  return (
    <div className="donudynomap-panel">
      <h3>Donudynomap Controls</h3>
      {/* Presets Section */}
      <div className="presets-bar">
        <strong>Presets: </strong>
        <button onClick={() => applyPreset(presets.spinningTop)}>Spinning Top</
button>
        <button onClick={() => applyPreset(presets.breathingPrecession)}
>Breathing Precession</button>
        {/* ...other preset buttons... */}

{/* Save Preset could be a button that captures current state (not fully
implemented) */}
        <button onClick={() => {/* TODO: save current preset */}} disabled>Save
Preset</button>
      </div>

      {/* Main Controls */}
      <div className="controls">
        <label>Axial Spin: <input type="range" min="-10" max="10" step="0.1"
                                   value={axialSpin} onChange={e =>
setAxialSpin(parseFloat(e.target.value))} /></label>
        <label>Poloidal Spin: <input type="range" min="-10" max="10" step="0.1"
                                      value={poloidalSpin} onChange={e =>
setPoloidalSpin(parseFloat(e.target.value))} /></label>
        <label>Toroidal Spin: <input type="range" min="-10" max="10" step="0.1"
                                      value={toroidalSpin} onChange={e =>
setToroidalSpin(parseFloat(e.target.value))} /></label>
        <label>Orbital Spin: <input type="range" min="-10" max="10" step="0.1"
                                     value={orbitalSpin} onChange={e =>
setOrbitalSpin(parseFloat(e.target.value))} /></label>
        <label>Phase-Lock Ratio:
          <input type="number" value={phaseLock.ratio} min="0.1" max="5"
step="0.1"
                 disabled={!phaseLock.enabled}
                 onChange={e => setPhaseLock({ ...phaseLock, ratio:
parseFloat(e.target.value) })} />
          <input type="checkbox" checked={phaseLock.enabled}
                 onChange={e => setPhaseLock({ ...phaseLock, enabled:
e.target.checked })} /> Enable
        </label>
        <label>Coherence: <input type="range" min="0" max="1" step="0.01"
```

```
                                   value={coherence} onChange={e =>
setCoherence(parseFloat(e.target.value))} /></label>
        <label>Emission: <input type="range" min="0" max="1" step="0.01"
                                  value={emission} onChange={e =>
setEmission(parseFloat(e.target.value))} /></label>
      </div>

      {/* Toggle for Advanced drawer */}
      <button onClick={() => setShowAdvanced(!showAdvanced)}>
        {showAdvanced ? 'Hide Advanced ▲' : 'Show Advanced ▼'}
      </button>

      {/* Advanced Controls Drawer */}
      {showAdvanced && (
        <div className="advanced-controls">
          <h4>Advanced Options</h4>
          <label>Shear: <input type="range" min="0" max="1" step="0.01"
                                 value={shear} onChange={e =>
setShear(parseFloat(e.target.value))} /></label>
          <label>Coupling Mode:
            <select value={couplingMode} onChange={e =>
setCouplingMode(e.target.value)}>
              <option value="uncoupled">Uncoupled</option>
              <option value="soft">Soft Coupling</option>
              <option value="hard">Hard Coupling</option>
            </select>
          </label>

{/* Additional advanced controls (e.g., damping, field effects) can be added
here */}
        </div>
      )}

      {/* Demo Mode Canvas */}
      {isDemo && (
        <div className="demo-canvas" style={{ border: '1px solid #555',
marginTop: '10px' }}>
          {/* If using react-three-fiber, we could embed a Canvas here */}
          {/* <Canvas style={{ width: '100%', height: '200px' }}>
                <ambientLight />
                <pointLight position={[5,5,5]} />
                <Torus args={[1, 0.3, 16, 100]} rotation={[0,0,0]}>
                  <meshStandardMaterial color="#6699cc" metalness={0.3}
roughness={0.6} />
                </Torus>
                <SpinningAnimation axes={{ axial: axialSpin, poloidal:
poloidalSpin, ... }} />
```

```
              </Canvas> */}
          <p align="center"><em>Demo mode:</em> Gyro is inactive. Showing
simulated motion.</p>
        </div>
      )}
    </div>
  );
}

export default DonudynomapPanel;
```

(The above code is illustrative: it outlines the component structure, state, and event handlers. In a real implementation, we would replace the simple input sliders with styled components or a library UI. The Three.js demo canvas inside the panel is shown conceptually with a `<Torus>`; we'd need to implement the rotation animation, e.g. using a custom hook or `useFrame` to rotate the torus mesh according to the state.)

Integration Notes:

- **Styling & Layout:** We would use CSS/JSX styling to ensure the panel is compact. For example, sliders could be narrow and perhaps horizontal with labels on the left. The advanced drawer might have a slightly indented background to distinguish it. We should also make sure the panel can fit into Membrane's UI without causing overflow – possibly by making it scrollable if too tall.
- **Performance:** Updates are real-time, so we must ensure the simulation can handle rapid changes. The `useEffect` approach will send a lot of updates if the user is dragging a slider. If needed, we could throttle the updates (e.g., update at 60Hz max) or let the simulation itself interpolate smoothly. Given the likely modest complexity of a spinning donut, direct updates on slider events should be fine.
- **Preset Management:** For now, the preset save button is disabled. In the future, implementing it might involve opening a modal to name the preset and storing the preset object in localStorage or in Membrane's backend. We have structured the state such that capturing it into an object and later re-applying it is straightforward (as shown in `applyPreset`). The concept mirrors how other UI tools allow state saving for controllers [4], so extending this panel with persistent user presets will be feasible.

By following this design and implementation plan, the **Donudynomap panel** will seamlessly integrate into Membrane, offering users an intuitive and powerful interface to manipulate the donut-gyro simulation. The combination of immediate visual feedback [2], organized basic vs. advanced controls [3], and helpful presets makes exploring complex gyro dynamics both accessible and engaging. The panel's code structure ensures maintainability and future extensibility (e.g., adding more controls or tying into other app systems), aligning with Membrane's modular panel system.

Overall, this proposal balances **usability** (clean default UI with progressive disclosure of complexity) and **functionality** (direct hooks into the Three.js simulation for real-time control, with a fallback demo). Users from novices to experts can experiment with gyro dynamics — from simply spinning a donut like a top, to crafting synchronized multi-axis motion patterns — all within this interactive Donudynomap panel. The result is a rich, responsive UI component that enhances the Membrane app's simulation experience.

Sources: The design takes inspiration from known patterns in 3D simulation UIs and control libraries, such as using collapsible sections for grouping advanced parameters [3] and enabling immediate updates to the scene when controls change [2]. Multi-axis rotation of a torus (donut) has been demonstrated in classic demos [1], informing how we conceptually map axial/poloidal/toroidal rotations in the UI. The preset saving approach is influenced by features in GUI frameworks (e.g., lil-gui) that allow capturing and restoring user settings [4]. These references guided the creation of a panel that is technically sound and user-friendly.

---

[1] Donut math: how donut.c works
https://www.a1k0n.net/2011/07/20/donut-math.html

[2] [3] Leva - React Three Fiber Tutorials
https://sbcode.net/react-three-fiber/leva/

[4] lil-gui 0.21.0
https://lil-gui.georgealways.com/