ChatGPT

Donut of Attention / Solar OS – Conceptual Framework and Developer Blueprint

Part A: Conceptual Report – Geometry, Fields, and Symbolic Mapping

Topological Design – The Toroidal Attention Manifold: The core of the **Donut of Attention** is a 3D torus (donut-shaped manifold) that serves as a "workspace" for attentional and cognitive states. A torus is mathematically suitable because it has two principal circular dimensions – ideal for mapping dual aspects of cognition (e.g. two concurrent focus variables, context vs. content, etc.) on a closed, continuous surface. In neuroscience, continuous **attractor networks** often manifest as rings or tori; for example, recordings from grid cells show that an animal's 2D spatial position maps onto a toroidal manifold of neural activity [1]. This means the brain itself may use torus-like geometry to encode continuous information. By leveraging a torus, we tap into a geometry the brain *already* finds natural for representing looping, periodic relationships (like cycles of attention or working memory loops). The torus's **looped topology** intuitively represents recurrent cognitive processes – an idea reinforced by Edelman and Tononi's concept of reentrant loops in the brain that integrate distributed neural activity into a coherent state [2]. Each "loop" around the torus can symbolize a **closed cycle of attention** or thought: as attention travels around a loop, it eventually returns to its starting point, suggesting a refresh or rehearsal of information (much like focusing, drifting, and refocusing on a topic). This loop intuition extends to **knotting** on the torus as well – stable patterns of attention could form knot-like trajectories (closed loops that intertwine without unraveling) indicating robust, recurring thought patterns or habits.

Conceptual models from recent research illustrate how a torus can serve as an information manifold. In panel A (left), a torus connects a "black hole" core (converging and compressing information) to a "white hole" outlet (unfolding and expanding information), visualizing a continuous enfolding–unfolding cycle of information flow. In panel D (right), various torus-shaped knots are depicted as standing-wave attractors – stable loops of information in the fabric of reality [3] [4]. These motifs inspire our design: the Donut's center could represent a well of concentrated insight (all inputs enfolded), while its surface projects those insights outward in accessible form (ideas unfolded into the UI). Recurrent knot-like patterns on the torus correspond to recurring cognitive loops (e.g. a repetitive thought or a learned routine), giving users a visual anchor for their stable mental routines.

Layered Fields – Mapping Cognitive Dynamics onto the Donut: We treat attention and working memory as **fields overlaying the toroidal surface**, continuously evolving like colored patterns on a sphere. Each field is essentially a distribution of some quantity (e.g. "attention intensity" or "working memory usage") across points on the torus. For example, a bright glow or heightfield on the torus surface could denote areas of focused attention, while cooler/darker regions indicate peripheral awareness. Multiple layers can coexist on the same torus: imagine one semi-transparent heatmap for the user's focus of attention, and beneath it another layer representing working memory activation. The design can use **toroidal coordinates** (angular dimensions) to partition different cognitive contexts – e.g. one angular direction might represent semantic categories or tasks, and the other represents time/sequence or priority. In this

way, the torus naturally encodes a **2D state-space of mind** (two key mental variables spanning the torus). Notably, a 2D torus is topologically a product of two circles ($S^1 \times S^1$); if more dimensions are needed (e.g. additional context), higher-dimensional tori or stacked rings could be used, but the UI will primarily visualize the 3D embedding for clarity.

Crucially, the fields on the torus obey **dynamical systems principles**. We can conceive of attention as a **continuous field that self-organizes** to highlight certain regions (attractors) while suppressing others, analogous to how a magnetic field might concentrate at certain poles. This dynamic follows concepts from Karl *Friston's free-energy principle*: the system tries to minimize surprise and maintain an optimal balance of attention [5]. In practice, this could mean the attention field automatically smooths and recenters itself to avoid over-concentrating on one point for too long (preventing "blow out") or spreading too thin (preventing waste) [5]. Peaks in the field correspond to "focus points" – maybe a task or concept the user is heavily concentrating on – while troughs indicate ignored or background matters. The field's evolution can be guided by an **order parameter** that reflects the level of focus vs. diffusion (high order = one sharp focus; low order = scattered attention). Thanks to the torus's wraparound, if a focus "peak" drifts off one edge, it seamlessly re-enters on the other side – a visualization of how our mind cyclically revisits topics. The system's dynamics might even have a Lyapunov-stable attractor state representing the **optimal attention distribution** for the current context, to which the field will gently return if perturbed (echoing the idea of a mind finding its equilibrium). Meanwhile, **working memory** can be visualized as another field or as discrete items (see symbolic anchors below) that orbit the torus. Research in RNNs suggests that storing multiple continuous items naturally leads to torus-like manifolds [6] [7] – this aligns with our approach of plotting memory items around loops on the torus to avoid interference. If needed, a higher-dimensional torus (Clifford torus in 4D) could underlie the simulation to keep certain streams orthogonal [7], but visually we'd still show a 3D projection.

We incorporate **multi-layer overlays** on this torus to represent different "subsystems" of cognition in a unified space. This is inspired by **integral/holonic theory** (Ken Wilber's idea that systems are composed of holons – each element is both a whole and a part of a larger whole [8]). In our design, each layer on the torus is a "whole" field (say, the field of attention) but also part of the larger holistic state (the entire cognitive context). The layers can be semi-transparent and stacked, or rendered as concentric translucent torus surfaces. A **boundary overlay** might be used to mark qualitative zones on the torus – for instance, dividing quadrants or highlighting a "ring" around the torus that represents a threshold between conscious focus and fringe awareness. These boundaries act like **isoclines** or separatrices in dynamical systems, indicating where a slight push could snap attention into a new regime. They might be drawn as glowing lines circling the donut (using the torus's circular symmetry to our advantage for easy visualization). The **implicate/explicate concept** from David *Bohm* provides a useful lens here: the torus's inner structure can be seen as the **implicate order** – a hidden layer where different cognitive elements are tightly **enfolded** together – while the surface is the **explicate order**, the visible manifestation that the user interacts with [9]. In other words, the system might maintain complex relationships and history (implicate, beneath the hood), but what the user sees on the donut's surface is a coherent projection of that complexity (explicate). This is analogous to Bohm's holographic metaphor where each region contains the whole: every part of the Donut's field could potentially encode a reflection of the entire mental state [9]. (In fact, neuro-holographic theories like Pribram's suggest each piece of the brain holds the memory of the whole [10] – our design plays with this idea by making even a small region of the torus informative of global state.) Practically, this could mean if the user zooms into a particular section of the torus, subtle patterns there echo the overall distribution – a fractal-like self-similarity that provides context at any scale.

Furthermore, drawing from **Integrated Information Theory (IIT)**, we consider **integration density** as a metaphorical metric on the torus. IIT posits that the quantity of consciousness (Φ) corresponds to how integrated a system's information is [11]. In the Donut UI, regions where multiple streams of information converge (e.g. where an attention peak overlaps with a memory item and a contextual boundary) could be highlighted or made more intense, indicating "high Φ" zones – points of especially rich, integrated meaning. For instance, if two ideas previously separate on the torus suddenly form a looped linkage (a knot), that could light up as an "integration event," symbolizing a moment of insight or synthesis. This is an optional layer of meaning, but it offers a **mythopoetic** interpretation of system state – the torus not only shows where attention is, but *how integratively* the mind is operating. High integration might be depicted by a luminous band wrapping around the torus (a "ring of light") or a change in the torus's resonance (perhaps a hum or vibration if audio is involved). By giving visual form to integration, we nod to Tononi's IIT in spirit, without claiming any literal measure of consciousness – it's a metaphor to guide users toward balancing differentiation and unity in their focus (too fragmented vs. too monolithic).

Symbolic Anchors and Mytho-Poetic Mappings: To make this abstract toroidal field intuitively meaningful, we attach **symbols, icons, and metaphors** at key locations – these are the *anchors* that tie the user's direct experience or tasks to the geometric model. Each anchor could be a small glyph or 3D icon perched on the torus surface (or slightly above it), representing a specific item: for example, a notepad icon for a note in working memory, a ⚡ lightning bolt for a sudden insight, an eye 👁 for an attention focal point, etc. These symbols serve as **semiotic bridges** between the user's mental content and the UI. As philosopher Susanne Langer noted, "symbols are not proxy for their objects, but are **vehicles for the conception** of objects" [12] – in our design, an icon on the Donut doesn't just mark a thing, it evokes the *concept* of that thing in the broader context of the user's mental state. By consistently using certain icons (mythical or archetypal where appropriate), we infuse the interface with a narrative layer. For instance, a **flame icon** might denote a topic the user is "passionate" about – the flame's location and brightness on the torus shows *where* and *how strongly* that passion is present in the attention field. Multiple flame icons clustering could signify a "burning issue" integrating into a larger fire (echoing the idea of many small concepts merging into one big insight).

We draw inspiration from **mythological and integral frameworks** to choose these mappings. The very shape of a torus invites the mythic symbol of the **Ouroboros** – the serpent eating its tail – which represents a self-contained cycle of renewal. The Ouroboros is essentially a ring (often drawn as a circle), but as a torus it gains volume: it can symbolize how the inner and outer, the self and environment, continuously transform into each other. In our interface, we might stylize the entire donut with subtle scale or feather textures, suggesting a serpent coiled in on itself, or we might simply use the Ouroboros as a guiding metaphor communicated to users (e.g. calling a full rotation of the torus an "Ouroboros cycle"). The **inside-to-outside mapping** of the torus (inner core vs outer surface) embodies what one might call a *cognitive twist* – a concept echoed by Meijer's description of the toroidal Ouroboros as relating inside to outside like a Möbius strip [13]. This metaphor reinforces to the user that what is deep inside their mind (inner core of torus) will eventually manifest externally (outer surface) and vice versa; ideas circulate. We can make this concrete by, say, placing more **intimate/personal symbols on the inner ring** of the torus (closer to the donut hole) and more **public/external symbols on the outer ring** – a kind of concentric meaning gradient.

We also consider **Ken Wilber's integral holonic model**: the idea that each element is a whole/part suggests a nested layering of meaning. We might represent this by **nesting smaller tori or circles within the main torus** (a torus of tori?). In fact, French psychoanalyst *Jacques Lacan* once modeled the human psyche's three registers – the Real, the Symbolic, and the Imaginary – using three interlinked torus shapes (a topological Borromean knot of three rings) [14]. Taking a cue from that, our design could have three

interlinked layers or bands on the Donut of Attention: one layer for raw sensory or "Real" inputs (e.g. an ambient field of what's being sensed), another for "Symbolic" processing (the user's language, symbols, tasks – the icons we place), and a third for "Imaginary" or visual thinking (perhaps a layer for daydreams, visualizations). These three layers would overlap and influence each other on the same torus, much like Lacan's rings knot together such that removing one makes the whole structure fall apart. This is an optional depth for users who appreciate a rich metaphorical framing; it can be communicated in onboarding that the Donut isn't just a random shape – it's explicitly representing **mind as a holarchy of intertwined domains** (each with its own flavor of content). Concretely, the user might toggle view modes to emphasize one layer (e.g. a "symbol layer view" shows the icons clearly, a "visual layer" might show more nebulous imagery on the torus for imagination, etc.), or see them all at once in composite.

Each symbolic anchor on the torus is carefully chosen to leverage **cultural and intuitive semantics**. We might borrow from C. S. Peirce's triadic sign theory: some anchors will be **iconic** (visually resembling what they mean, like a picture of an ear on the torus for listening mode), some **indexical** (located in a way that points to context, like a flag marking the "north" of the torus meaning highest priority), and some **symbolic** in the strict sense (arbitrary but agreed-upon, like a rune or abstract shape whose meaning the user learns). This mix ensures the interface is immediately graspable (through icons) yet also deep and customizable (through learned symbols). We also integrate **semiotic consistency**: colors, shapes, and placement all carry meaning. For example, **color gradients** on the torus might correspond to emotional tone (a red hue field in a region could mean urgency or stress in that part of attention, whereas blue could mean calm or idle thoughts). The *Solar OS* theme itself encourages a cosmological aesthetic: the torus might be textured subtly like a solar surface or magnetic field lines, and icons could have a celestial motif (e.g. planets for major projects orbiting the sun-like center of attention). Indeed, we might conceptualize the **center of the torus as a sun** (hence "Solar OS") – the shining core of awareness – and the torus as a halo or orbiting **accretion disk** of thoughts around that sun. This links to *Bohm's* idea of the holomovement where everything unfolds from a unified whole [15] [16]. The holographic angle (inspired by *'t Hooft's holographic principle and Bohm) we incorporate by ensuring that information on the torus's surface is richly representing the hidden content**. Just as in a hologram each piece contains the whole image, here any given region of the donut can be "read" to infer the general state (with practice). For instance, the user might learn that when their torus shows a certain spiral pattern on one side, it always means they are in a particular mental mode (because the whole field shifts holistically during that mode).

In summary, the Conceptual Framework brings together **geometry (toroidal manifold)**, **dynamics (fields, attractors, free-energy minimization)**, and **meaning (semiotic anchors, mythic metaphors)** into one unified construct. The torus is not just a pretty shape – it's a functional map of mind, where *topology meets cognition*. Drawing on lenses like holonics, holographics, active inference, IIT, and semiotics, we ensure every design choice has a rationale: e.g. using a torus because the brain does (at least in grid cells) [1], overlaying fields because cognition is continuous, employing loops and knots because thoughts recur and entangle, and using symbols because users need familiar touchpoints in an abstract space. Each theoretical lens is tied back to UI choices: **Integral theory** gives us layered/contextual views [8], **Bohm's implicate order** gives us the idea for an inner hidden state vs outer display [9], **Friston's principle** informs the self-stabilizing field behavior [5], **Tononi's IIT** inspires visualizing integration intensity [11], **Smolin's loops** (loop quantum gravity) inspire representing connections as networks of loops [17], and **Langer/Peirce's semiotics** ensures the user sees not just data but *meaning* [12]. The result is a "**Solar OS**" interface that feels alive, holistic, and resonant: a toroidal hologram of the user's attention, where abstract data is transformed into a living landscape of lights, symbols, and flows that they can navigate and manipulate.

Part B: Developer Blueprint – Implementation in a Web-Native Stack

Technology Stack Overview: The implementation will use a **web-first approach** to ensure portability and easy integration with the existing codebase (which we assume already uses web technologies, given the preference for plugging into Donut's current stack). We recommend using **Three.js** (a popular WebGL2 framework) as the 3D engine to render the torus and its dynamic fields. Three.js provides high-level abstractions for geometry, materials, shaders, and scene management, which accelerates development. For the UI scaffolding around the 3D scene, a modern reactive framework like **React** (with something like ReactDOM or even React Three Fiber) or **Svelte** can be used – this will handle any 2D interface elements, state management, and integration of the 3D view into a larger application. The build system can be **Vite** (a fast modern bundler) to manage dependencies (like three.js modules, GLSL shader files, etc.) and enable hot-reloading during development.

This setup means our **Donut of Attention** can run in any browser, and easily be deployed as part of a web app or an Electron cross-platform app. WebGL2 is sufficiently powerful for complex shaders and 3D shapes, and if in the future **WebGPU** becomes more broadly supported, we can consider migrating performance-critical parts (like heavy field simulations) to WebGPU for compute shaders or more advanced graphics. However, the current design can be fully realized with WebGL2 + Three.js. (Notably, Three.js has an experimental WebGPU renderer, so there's a pathway to upgrade when ready – but we'll stick to WebGL2 for stability).

3D Scene and Torus Setup: We begin by constructing the torus in Three.js. Three.js conveniently has a `TorusGeometry` which we can use to create the donut shape. We'll choose parameters for the torus such that it provides a smooth surface for our fields: e.g. `TorusGeometry(radius=1, tubeRadius=0.3, radialSegments=64, tubularSegments=128)`. This gives a moderately high-poly torus (8192 faces) which is fine for modern GPUs and ensures our shader visuals look smooth (no chunky facets). The torus will be added to a `Scene`, and a `PerspectiveCamera` will view it – likely positioned slightly off-center so the user sees the donut at an angle (e.g. camera at [x=2,y=2,z=2] looking at origin gives a nice 3D view of the torus). We'll enable user controls for orbiting the camera – Three.js's `OrbitControls` can allow click-and-drag rotation of the scene so the user can turn the donut to inspect all sides. We might constrain the orbit so the camera doesn't flip upside down (to keep "up" consistent, unless we intentionally want a fully free rotation). Lighting will be simple: perhaps an ambient light for overall illumination and a directional light to create some shading on the torus, which can help cues of depth. However, since much of the torus's appearance will come from our custom shader (which can handle lighting calculations or emissive glows itself), we might not rely heavily on Three.js lighting. We can set the torus material to a **ShaderMaterial** (see next section) for full control of its pixel coloring.

Other scene elements: if we want a background, we can add a subtle starfield or gradient sky to fit the Solar OS theme (Three.js can set a scene background or we could use a large sphere with a space texture). This is aesthetic; the main focus is the torus. We will also create objects for the symbolic anchors: for each icon or marker, a simple approach is to use a `Sprite` or a flat `PlaneGeometry` with a texture (icon image). Three.js Sprites always face the camera, which is good for 2D icons in 3D space. Alternatively, for richer detail, we could use small 3D models or text labels (Three.js has TextGeometry or we can use CSS2DRenderer for HTML-based labels). Each anchor will be positioned at a point on or slightly above the torus surface – to compute that position, we convert the torus parameter coordinates (two angles **u,v**) to

XYZ coordinates on the torus. (Three.js `TorusGeometry` can give us the vertex positions, or we can do our own math: torus param eq: `pos(u,v) = [(R + r*cos(v)) * cos(u), (R + r*cos(v)) * sin(u), r * sin(v)]` for major radius R and tube radius r.) Storing those param coords for anchors means we can animate them or recompute if the torus deforms or if an anchor needs to slide along the surface.

Custom Shader for Fields: To visualize attention and other fields on the torus, we will write GLSL shaders. Three.js allows us to supply a custom `vertexShader` and `fragmentShader` via a `ShaderMaterial`. We'll take advantage of this to implement things like heatmaps, pulsating patterns, or procedural textures on the torus. The shader will receive uniform data representing the current state of the attention field. We have a couple of options for feeding the field data: (1) as a **texture** (e.g. a 2D texture where U,V correspond to torus coordinates and the pixel values encode intensity), or (2) as a mathematical function coded in the shader (if the field can be described analytically or via noise). The texture approach is more general – we can generate a 2D array (say 256×256) of values on the CPU (or even GPU via Framebuffer if dynamic) representing attention intensity across the torus surface, then update that texture each frame or as needed. The fragment shader would sample this texture with the fragment's torus coordinates to determine its color. The coordinate mapping is straightforward since the torus has a well-defined UV parameterization.

For example, if using Three.js's UVs: a torus geometry by default provides UV coordinates that map the donut's surface to a square image. We must confirm how Three.js lays out the UV (likely 0–1 along the tube and 0–1 along the ring, straightforward). We might set the texture wrap mode to Repeat to ensure continuity along the seams. If generating the field texture, we'll update its pixels based on the app's state: e.g. if the user's focus shifts, the "peak" in the texture moves accordingly (we could simulate a Gaussian or bump moving). This can be done each animation frame or at some lower frequency if it's expensive. Alternatively, for smoother animation, we can encode rules in the fragment shader: e.g. treat certain anchor points as sources of "attention energy" and calculate intensity as a function of distance from those points on the torus. This would let the GPU compute the field on the fly without texture updates. For instance, if we have N focus points, we pass their positions as uniform arrays and in the shader sum up contributions like `float intensity = 0.0; for(int i=0; i<N; ++i){ float d = distance(surfaceCoord, focusPoints[i]); intensity += exp(-d*falloff); }`. This could create multiple gaussian blobs of attention on the torus. The choice might depend on number of points and flexibility – a texture can encode any shape (even from real data or complex simulation), whereas analytic might be simpler for a few points.

The shader will output color and possibly some **emissive glow**. We might decide a certain baseline color for the torus (e.g. a deep blue or black, if we want it mostly dark except where attention highlights it). Then add the field intensity as a glow map – e.g. orange/yellow for high attention zones, fading to transparent in low zones. If using a heatmap, we can have a predefined gradient (blue->green->yellow->red or such) and sample it based on intensity. This can be done by another 1D colormap texture or by logic in shader. For instance: `if(val < 0.5) color = mix(blue, green, val*2.0); else color = mix(green, red, (val-0.5)*2.0);`. High values might even bloom using a post-processing effect (Three.js has bloom filters) to really emphasize intense focus points. For **temporal dynamics** (like pulsing or oscillating fields), we add a uniform `time` (updated each frame) and use it to animate. Example: we could make the attention peak throb subtly by `val = baseIntensity * (1.0 + 0.1*sin(2.0*pi* time * frequency))`. Or implement a rotation of the field (like if we want the whole pattern to slowly drift around the torus to indicate flow of time or something). Because a torus is symmetrical, we could even spin the texture coordinates over time for a conveyor-belt effect – but careful, that might confuse meaning unless it's deliberate (perhaps representing the passage of time as a rotation of the torus's field).

Below is a pseudocode snippet illustrating creation of the torus and a shader material in Three.js, including how we might pass uniforms:

```
// Pseudocode: Setting up Three.js scene with a torus and custom shader
const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera(75, window.innerWidth/
window.innerHeight, 0.1, 1000);
camera.position.set(2, 2, 2);
camera.lookAt(0, 0, 0);
scene.add(new THREE.AmbientLight(0xffffff, 0.5));
const light = new THREE.DirectionalLight(0xffffff, 0.8);
light.position.set(5,5,5);
scene.add(light);

// Create torus geometry and shader material
const torusGeom = new THREE.TorusGeometry(1, 0.3, 64, 128);  // (R=1, r=0.3)
const shaderMat = new THREE.ShaderMaterial({
  vertexShader: torusVertexShaderSource,
  fragmentShader: torusFragmentShaderSource,
  uniforms: {
    time: { value: 0.0 },
    fieldTexture: { value: generateInitialFieldTexture() },  // if using texture
    focusPoints: { value: initialFocusPointsArray },         // if using
analytic points
    // ... other uniforms like colors, scaling factors
  },
  transparent: true,   // allow glow or transparency if needed
  side: THREE.DoubleSide // render both inside/outside surfaces if needed
});
const torusMesh = new THREE.Mesh(torusGeom, shaderMat);
scene.add(torusMesh);

// Animation loop (e.g., using requestAnimationFrame or Three.js clock)
function animate() {
  requestAnimationFrame(animate);
  // update time uniform for animations
  torusMesh.material.uniforms.time.value = performance.now() / 1000;
  // optionally update field texture or focusPoints uniforms based on app state
  updateAttentionField(torusMesh.material.uniforms);
  renderer.render(scene, camera);
}
```

In this snippet, `torusVertexShaderSource` and `torusFragmentShaderSource` are strings containing GLSL code. The **vertex shader** can be relatively basic – just pass through the position and maybe compute or pass the UV coordinates or any other necessary data to fragment. The **fragment shader** is where the

magic happens: it will sample `fieldTexture` at `uv` (or compute intensity from `focusPoints`) and then output a color. For example, a simple fragment shader might do:

```
uniform sampler2D fieldTexture;
uniform float time;
varying vec2 vUv;
void main() {
    float intensity = texture2D(fieldTexture, vUv).r;    // sample red channel
as intensity
    // Apply a dynamic effect, e.g., make it pulse with time:
    intensity += 0.1 * sin(time * 6.283); // small oscillation
    intensity = clamp(intensity, 0.0, 1.0);
    // Map intensity to a color (blue to red for example):
    vec3 baseColor = mix(vec3(0.0,0.0,1.0), vec3(1.0,0.0,0.0), intensity);
    gl_FragColor = vec4(baseColor, 1.0);
}
```

This would color the torus from blue (no attention) to red (full attention) with a slight pulse. In practice, we'll use a more nuanced palette and maybe add an emissive bloom for the brightest spots.

Data Binding and Interaction Patterns: The next aspect is how to drive this visualization with real data and make it interactive. We assume there is some underlying data (possibly from the user's context or a cognitive model) that provides the positions or intensities of attention and memory items. These will be fed into the visualization in real-time. For instance, if the user opens a new task, we might get a new "focusPoint" to display, or an existing point's "intensity" might increase. We'll design the system such that updating the visualization is straightforward: e.g. a React state or Svelte store holds the current state of attention (maybe as an array of objects like `{id, angleU, angleV, intensity, symbolType}` for each item). Whenever this state updates, we trigger a corresponding update in Three.js: move or change an anchor sprite, update the field texture or uniforms, etc. Because Three.js objects are long-lived, we ideally initialize them once and then just change their properties each frame or on events (rather than rebuilding geometry). This is where using a reactive library with something like **react-three-fiber (R3F)** could shine: R3F lets us treat Three.js components as part of React JSX, binding their properties to state declaratively. For example, with R3F, we could do something like:

```
<mesh geometry={torusGeom} rotation={[0,0,0]}>
  <shaderMaterial ref={matRef} uniforms-... />
</mesh>
{focusItems.map(item => (
  <sprite key={item.id} position={toTorusPosition(item.angleU, item.angleV, 1,
0.3)}>
    <spriteMaterial map={iconTextures[item.symbolType]} />
  </sprite>
))}
```

Here, any change in `focusItems` array (state) would add/remove/move sprites accordingly. If not using R3F, we can manually manage this: e.g. in the main JS, loop through current items, and for each anchor object in the scene update its `position.set(x,y,z)` using the torus param conversion (with some easing perhaps for smooth motion).

Interactivity includes **user input**: The user should be able to rotate and zoom the torus (OrbitControls already covers rotate; we can allow zoom via scroll to let them inspect details or see the whole donut). Clicking or tapping on a symbolic anchor should bring up detail – this can be handled via **raycasting** in Three.js (casting a ray from camera through mouse position to find intersected objects). When an anchor (sprite or mesh) is clicked, we can trigger a callback that, for example, opens a React sidebar or overlay with information about that item (e.g. "Task: Write report, Deadline: tomorrow"). This means our anchor objects should have identifiers or references back to the actual data (we can set `sprite.userData = { itemId: ... }` to know what it represents). Three.js Raycaster gives us the object, then we lookup its userData to identify it. This click handling can be integrated with React by dispatching an event or using R3F's built-in event support (sprites can have `onClick` prop in R3F).

Additionally, the user might interact directly with the fields – for example, maybe by **drawing on the torus** or **nudging a focus point**. We could allow drag of an anchor: on pointer down, attach to the pointer and as they move, recompute the underlying angles from the 3D position and update the data (this is more advanced, but doable with some spherical->toroidal coordinate math and maybe constraining movement along the surface). Another possibility is allowing the user to "paint" attention on the torus (like scribble to highlight something they want to focus on) – this would require writing into the field texture via an interaction (capturing pointer movement and translating to UV coordinates, then boosting those in the texture). This is a stretch goal, but shows the potential for a creative, immersive UI.

System Components Breakdown: To organize development, we can break the blueprint into modular components:

- **TorusCanvas Component:** A component (could be React, Svelte, or vanilla JS module) that initializes the Three.js scene, camera, controls, and renderer. It also houses the torus mesh and manages the animation loop. It exposes an API to update the field or anchors (or directly ties into app state).
- **AttentionField Shader:** The GLSL code + logic for updating its data. This might be stored in a separate `.glsl` file for maintainability. We might create a small utility to generate field textures or to compute intensity given state.
- **SymbolAnchor Manager:** A part of the code responsible for creating and updating the symbolic anchor sprites/meshes. This includes loading any icon textures (we'll have a set of PNG/SVGs for icons like flame, eye, etc.), and positioning them correctly. We can also manage showing/hiding them (e.g. if an item is not currently in working memory, its icon might fade out or be removed).
- **Interaction Handlers:** Code that handles user input – e.g. an onClick on the canvas or pointer events – hooking into the raycaster and determining what was picked. This can be part of TorusCanvas or separate if using a framework's event system (R3F merges it, but with plain Three we manually add `renderer.domElement.addEventListener('pointerdown', ...)` etc.).
- **UI Overlay Components:** Standard HTML/CSS/React/Svelte components for things like info panels, tooltips on hover (e.g. hovering an anchor could show a tooltip with its name), toggle buttons for different layers (Real/Symbolic/Imaginary or different field views), and maybe a minimap or legend explaining colors. These will coordinate with the 3D scene (e.g. clicking a legend could adjust a uniform in the shader to change color scheme, etc.).

Performance considerations: The scene described is not extremely heavy – a ~8k poly torus, maybe a dozen sprites, and some fairly simple shaders – which should run comfortably at 60fps on typical hardware. If we start adding more (like hundreds of anchors or very large textures), we may need optimizations. For example, if the attention field is very dynamic and complex, updating a 256x256 texture every frame is fine (that's 65k updates, negligible), but if it were 1024x1024 maybe we'd do it at, say, 20fps and interpolate. Using the GPU to calculate the field (via shader) means we offload work from the CPU and can handle many points easily, but extremely many (hundreds) might need loops in shader – WebGL can handle loops but too many might drop performance. We could use a different strategy for many influences (like render each influence to an offscreen buffer additively). These are edge cases; at first, simplicity is key. We will also ensure to utilize Three.js's optimizations – e.g. mark things static if they are (the torus geometry doesn't change shape, so it can be static; only its material uniform changes). We'll leverage the **GPU** for all the heavy lifting of rendering and animating the attention field.

If down the line we want to incorporate **VR/AR (XR)**, Three.js has built-in WebXR support. We could let the user put on a VR headset and with one line `renderer.xr.enabled = true; renderer.setAnimationLoop( animate );` and some user input handling, they could literally walk around the Donut of Attention in a room. The design of using a real 3D torus pays off here – it's inherently suited for parallax and room-scale exploration if needed. For AR, using something like WebXR with AR mode or libraries like AR.js would allow placing the torus in the real world via a phone camera. While this is beyond the initial scope, we mention it to highlight that the **web-native approach keeps these options open** without needing to rebuild the logic. If native performance or specific platform integration becomes crucial, the same conceptual blueprint can be ported: Unity or Unreal could render a torus with a similar shader. Since we're using standard concepts (geometry, fragment shaders, etc.), replicating them in Unity's shader language (HLSL or ShaderGraph) or Unreal's Material editor is feasible. The **data flow and architecture would remain the same**, just with different APIs. But as requested, we emphasize web-first: the idea is the Donut of Attention lives as a component that can plug into web dashboards or desktop via Electron, ensuring maximum accessibility and ease of iteration.

Sample Code Snippet (integrated example): Below is a concise example tying a few pieces together – it creates the torus and an example anchor, updates the attention field, and handles a click on the anchor:

```
// Assume we have Three.js scene, camera, renderer set up...
const torusGeom = new THREE.TorusGeometry(1, 0.3, 64, 128);
const attentionShader = new THREE.ShaderMaterial({
  vertexShader: /* glsl */`
    varying vec2 vUV;
    void main() {
      vUV = uv;
      gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
    }
  `,
  fragmentShader: /* glsl */`
    uniform sampler2D fieldTex;
    uniform float time;
    varying vec2 vUV;
    void main() {
```

```
      float intensity = texture2D(fieldTex, vUV).r;
      // make intensity oscillate a bit with time
      intensity += 0.05 * sin(time * 6.283);
      intensity = clamp(intensity, 0.0, 1.0);
      // Map to color (e.g., black->blue->cyan->white as intensity grows)
      vec3 col = mix(vec3(0.0), vec3(0.0,1.0,1.0), intensity);
      col = mix(col, vec3(1.0), intensity * 0.5);
      gl_FragColor = vec4(col, 1.0);
    }
  `,
  uniforms: {
    fieldTex: { value: createFieldTexture(initialAttentionMatrix) },
    time: { value: 0.0 }
  }
});
const torus = new THREE.Mesh(torusGeom, attentionShader);
scene.add(torus);

// Create an example anchor sprite
const texLoader = new THREE.TextureLoader();
const ideaIcon = texLoader.load('icons/lightbulb.png');
const spriteMat = new THREE.SpriteMaterial({ map: ideaIcon, depthTest: false });
const ideaSprite = new THREE.Sprite(spriteMat);
// Position sprite at parametric coord (u=45°, v=30° as an example)
const u = Math.PI/4, v = Math.PI/6;
const R=1, r=0.3;
ideaSprite.position.set(
  (R + r*Math.cos(v)) * Math.cos(u),
  (R + r*Math.cos(v)) * Math.sin(u),
   r * Math.sin(v)
);
ideaSprite.scale.set(0.2, 0.2, 0.2);  // size of icon
scene.add(ideaSprite);
ideaSprite.userData = { name: "Idea: Learn GLSL Shaders" };  // attach info

// Raycaster for interaction
const raycaster = new THREE.Raycaster();
const pointer = new THREE.Vector2();
function onPointerClick(event) {
  // get pointer in normalized device coords (-1 to +1)
  pointer.x = (event.clientX / window.innerWidth) * 2 - 1;
  pointer.y = - (event.clientY / window.innerHeight) * 2 + 1;
  raycaster.setFromCamera(pointer, camera);
  const picks = raycaster.intersectObjects([ideaSprite]);
  if(picks.length > 0) {
    const obj = picks[0].object;
    alert("Clicked: " + obj.userData.name);
    // In practice, trigger UI panel or callback instead of alert.
```

```
  }
}
renderer.domElement.addEventListener('click', onPointerClick);

// Animation loop to update time uniform and re-render
function animate() {
  requestAnimationFrame(animate);
  torus.material.uniforms.time.value = performance.now()/1000;
  // Optionally update fieldTex if data changed...
  renderer.render(scene, camera);
}
animate();
```

This snippet is simplified (e.g., it only tests clicking one sprite), but it shows how the pieces come together: we create the torus with a shader, we create a sprite for a symbolic anchor, we position it via torus math, and we set up a click handler to interact. In a full application, we'd generalize this (handle many sprites, possibly using an array of objects and intersectObjects on all anchor objects, etc.), and we'd integrate it with the app's state management (replacing hardcoded positions with data-driven ones).

Integration with Existing Codebase: Given that the deliverable should plug into the Solar OS's Donut codebase, we'll ensure our blueprint aligns with their likely structure. If the existing codebase is already using React, we integrate by creating a new component (e.g. `<AttentionDonut3D />`) that encapsulates all the Three.js logic internally (perhaps using `useEffect` to set up the scene on mount, and cleaning up on unmount). The component would accept props like `attentionData` and `memoryData` which it uses to update the visualization. If the codebase is using Svelte, similarly we can create a Svelte component with onMount for Three.js init. The key point is that our system is **modular**: it doesn't impose a global state; it can take input from whatever higher-level state management exists in Solar OS (be it Redux, Context, Svelte store, etc.). For example, if Solar OS has a central model of user attention, we subscribe to that and in the callback update the Three.js uniforms accordingly. Also, any user interactions (like clicking an anchor) should dispatch actions or events back to the app (e.g. opening detailed view) – we will wire those through the framework's event system (like calling a provided `onSelectItem(itemId)` prop).

Portability considerations: While our primary target is WebGL in a browser, we acknowledge that at some point a native AR/VR deployment might be desired. The design choices here facilitate that: by using Three.js (which can export to WebXR) and by structuring the logic in terms of clear geometry and shader definitions, we could reimplement in Unity fairly directly. For instance, Unity could use a Torus mesh (primitive or imported) and we'd write a Unity shader graph replicating our fragment shader (texture sampling and coloring). Unity's XR plugins would handle VR device output. The **concept** of the torus and the mapping of data to visuals remains the same, so all the heavy thinking (done now) would be reused. Essentially, the blueprint ensures we're not locked into any proprietary tech – web-first is both the end product and a prototyping platform for any future expansions.

Summary of Developer Steps:

- *Step 1:* **Set up Three.js scene and camera**, ensure it renders to a canvas in the UI framework.
- *Step 2:* **Create the Torus mesh** with sufficient segmentation for smooth visuals.

- *Step 3:* **Write GLSL shaders** implementing the attention field visualization (and any other field layers). Bind uniforms for dynamic data (time, field maps, etc.).
- *Step 4:* **Load and place symbolic icons** as Sprites or meshes on the torus. Use meaningful positions (perhaps initially random or based on some default layout if no real data yet) and ensure they visually stand out (maybe give them a slight outline or glow so they don't get lost against the torus colors).
- *Step 5:* **Implement interactions:** camera controls for basic navigation, raycast picking for clicking anchors, possibly hover effects (e.g. change sprite scale or color on hover to indicate interactivity).
- *Step 6:* **Connect data:** hook up the actual attention model – whenever attention or working memory data updates, translate that into shader uniform updates (or texture updates) and anchor movements. Likewise, when user interacts (clicks an anchor), feed that back to open detail views or trigger state changes (e.g. marking an item as done could remove its icon and cause the field to redistribute).
- *Step 7:* **Testing and Iteration:** Adjust field visual mapping to ensure it's intuitive (we might tweak the color gradient or the scale of how intensity is computed so that the differences are visually clear). Also adjust performance settings (e.g. level of detail) so that even on lower-end devices it runs smoothly.
- *Step 8:* **UI polishing:** add any explanatory UI like a legend or guide, and ensure the component can resize with the window (listen to resize events and update renderer/camera aspect).

By following this blueprint, developers can build the **Donut of Attention** as a rich, interactive holographic donut that embodies the complex ideas of cognitive fields and mythic mappings, using solid web tech and informed by the cutting-edge conceptual lenses we've discussed. The end result will be an **experiential interface**: users can *see* and *play with* their attention as a living system – rotating a torus of thoughts, seeing their focus glow like a sunrise on the horizon of their mind, and recognizing patterns (the "knots" and "flows") in their cognitive habits, all within a responsive web-based application.

---

1 Toroidal topology of population activity in grid cells | Nature
https://www.nature.com/articles/s41586-021-04268-7?error=cookies_not_supported&code=28b413e6-ab8e-403f-88ae-f93a0ef1ddf4

2 Reentry (neural circuitry) - Wikipedia
https://en.wikipedia.org/wiki/Reentry_(neural_circuitry)

3 4 13 14 (A) Torus as a dynamic model for the recreation (rebirth) of our... | Download Scientific Diagram
https://www.researchgate.net/figure/A-Torus-as-a-dynamic-model-for-the-recreation-rebirth-of-our-Universe-from-a-wormhole_fig4_326972894

5 Friston's Free Energy Principle and Active Inference : r/neuroscience
https://www.reddit.com/r/neuroscience/comments/k9svca/fristons_free_energy_principle_and_active/

6 7 columbia.edu
https://www.columbia.edu/~nq6/publications/rnn-memory.pdf

8 The Stone Integral Matrix v0.9 - A Fork of Wilber's AQAL - Academy - Integral Life Community
https://community.integrallife.com/t/the-stone-integral-matrix-v0-9-a-fork-of-wilbers-aqal/662

9 15 16 Did Kabbalah Anticipate the Physics of David Bohm? - Part 2 of 3-part series - Chabad.org
https://www.chabad.org/library/article_cdo/aid/5546158/jewish/Did-Kabbalah-Anticipate-the-Physics-of-David-Bohm.htm

10 Implicate and explicate order - Wikipedia
https://en.wikipedia.org/wiki/Implicate_and_explicate_order

11 Integrated information theory - Wikipedia
https://en.wikipedia.org/wiki/Integrated_information_theory

12 Semiotics, Reality, and User Experience| Fusionbox Blog
https://www.fusionbox.com/blog/detail/semiotics-reality-and-user-experience/553/

17 Loop quantum gravity - Wikipedia
https://en.wikipedia.org/wiki/Loop_quantum_gravity