ChatGPT

Interactive Holographic and Tensor-Network Overlays in WebGL/Three.js

Visualization Techniques for Holography, Tensor Networks, and Platonic Structures

3D Geometric Overlays: A variety of WebGL/Three.js demos and libraries showcase complex geometric structures that could inspire DonutOS overlays. For example, interactive Platonic solid visualizations are readily available – from simple Three.js scenes of the five Platonic solids to more advanced combinations. Educational tools often depict a Platonic solid and its dual simultaneously (e.g. an octahedron inside a cube) to illustrate duality [1]. In Three.js, Platonic solids can be rendered easily (Box for cube, Tetrahedron, Octahedron, Dodecahedron, Icosahedron geometry), and one can overlay one shape within another to show correspondences (vertices of one at face-centers of the other) [2]. An interactive polyhedra viewer like **polyhedra.tessera.li** lets users manipulate numerous polyhedra in 3D [3] [4], and even highlights families (e.g. Archimedean, Johnson solids) built by augmenting or “gluing” simpler polyhedra [5]. Another excellent prototype is Patrick Kalita’s **Polyhedra Folding** demo, which **animates the folding/unfolding of polyhedral nets** – users can see a 2D net of a Platonic solid morph into the 3D solid and back [6]. This WebGL demo (built with Three.js) provides a clear pedagogical overlay: the flattened net (2D) and the fully folded 3D shape, illustrating how DonutOS could support **flattened 2D nets** alongside 3D forms. The demo uses geometry data from the Netlib Polyhedron Database and Three.js for rendering [6], showing that **implementation-ready patterns for polyhedral nets** exist.

Hyperbolic and Holographic Visualizations: To convey “fractal–holographic” concepts, one promising technique is using **Poincaré disk tilings** and hyperbolic geometry overlays. For instance, Malin Christersson’s interactive **Poincaré disk tiling** lets users *“Drag the white dots! Choose rendering style! ... Pick p and q!”* to explore regular hyperbolic tilings inside a disk [7]. This kind of demo (often done in Canvas or WebGL) visually represents an *infinite, self-similar pattern* – a clear nod to holographic or fractal structure – in a pedagogically approachable way. By adjusting parameters (p = polygon sides, q = polygons meeting at each vertex), users can see different tessellations of the hyperbolic plane. The recursive, circle-infinity nature of the Poincaré disk could be layered in DonutOS as a **background overlay** to suggest holographic or multi-scale relationships. In the physics realm, tensor network representations of the AdS/CFT holographic principle often use hyperbolic tilings (e.g. the classic “pentagon tiling” for a holographic code). While those are mostly static illustrations in research papers, one can imagine a Three.js scene mapping a 3D holographic bulk (like a warped 3D space) with a 2D boundary net – effectively a **Platonic or hyperbolic overlay** that users can toggle for understanding the “inside-outside” mapping. *Implementation tip:* three.js can handle non-Euclidean geometries via shader distortion or by mapping coordinates to a disk; even if true hyperbolic metrics aren’t native, visually approximating the tiling (as done in Malin’s demo) provides an interactive pedagogical tool.

Tensor Network Diagrams: For visualizing **tensor networks** (such as MERA or other entanglement networks) in-browser, there are GUI frameworks that prefer clarity. **GuiTeNet** is an open-source GUI (built with D3.js) for constructing arbitrary tensor network diagrams [8]. It represents tensors as nodes with "legs" (edges) and lets the user interactively connect them to define contractions [9] [10]. This means one can build a MERA graph by placing tensors (nodes) in layers (a multiscale hierarchy) and connecting indices – a very pedagogical approach to a complex concept. The interface supports dragging out new tensor nodes and attaching legs via drag-and-drop, then snapping leg tips together to **contract** tensors (with a "Contract" action) [9] [10]. As the user performs operations (adding tensors, contracting, splitting via SVD, etc.), the tool even generates live code (e.g. NumPy `einsum` calls) for those operations [11] [12]. While GuiTeNet itself is 2D (focused on diagram clarity), its approach could inspire a WebGL adaptation: *e.g.* a Three.js scene where tensors are small panels or spheres in 3D space and connections are drawn as tubes or lines. Importantly, GuiTeNet's design emphasizes clarity – every leg is labeled and color-coded, ensuring the **graph structure is understandable at a glance** [13] [14]. This matches DonutOS's goal of layered, pedagogical overlays. Even if DonutOS remains 3D, one could overlay a planar network diagram on a curved surface (like mapping a MERA network onto a hyperbolic disk, reflecting the known MERA–AdS connection). Libraries like **TensorSpace.js** take another approach: they visualize *neural networks* in 3D with Three.js, which is analogous since those are graph structures too. TensorSpace provides a high-level API to load pre-trained models and then generates a 3D scene of the network's layers; users can inspect how data flows through layers with animations and highlighted activations [15] [16]. This demonstrates an **implementation-ready pattern for showing information flow**: using color and motion to indicate values moving along connections in a network. Adapting this to tensor networks or quantum circuits is feasible – e.g. using animated pulses or gradients along an edge to show "contraction happening" or highlighting a specific tensor in the network to show a focus of computation. Overall, by combining these techniques, DonutOS can achieve rich **visual overlays**: imagine a torus (Donut) with a semi-transparent MERA network net draped over it, or Platonic solids floating around it as "navigation shells", all rendered with Three.js and toggle-able for the user.

Interactive Modeling: Spin Labels, Holonomy, and Compositional Overlays

Spin and Quantum Visualizations: Interactive graphics have been used to model quantum "spin" and related concepts in an intuitive way. A great example is the **Bloch sphere simulator**, which provides a 3D sphere representing a qubit's state and lets users apply rotations or gates to see the state vector move on the sphere [17]. Tools like IQM's Bloch applet or others on Observable allow dragging sliders or clicking on preset operations, with the vector updating live. This demonstrates how **spin labels** (in this case, the Bloch vector components) can be manipulated directly in a visual overlay. For DonutOS, if "spin labels" refers to, say, spins on the nodes of a network or lattice, a similar approach can be used: represent each spin by an arrow or orientation that the user can click/drag to flip. In Three.js, one could have small arrows or cones on a lattice that respond to user input and maybe change color (e.g. up vs down). The key is giving immediate visual feedback to reinforce the concept. For instance, a user could tweak a spin and an underlying tensor network overlay updates to show how that perturbation propagates (visualizing **information flow**).

Holonomy and Curvature Simulations: Visualizing **holonomy** (path-dependent rotations) and curvature is another rich interactive domain. A notable demonstration is the *Parallel Transport on a Sphere* animation by Nadeem Afana [18]. In it, a sphere rotates and vectors on its surface are parallel-transported along latitude circles; after a full loop around, the transported vector (green) is misaligned from the initial vector (blue), illustrating holonomic rotation due to curvature (the famous "triangle on a sphere" effect). As the blog notes, after going 2π around, *"the vector does not coincide with the original one… due to the curvature of* *the sphere."* [18] . This kind of animation could be made interactive in WebGL: users could drag the path of the vector or change the latitude and see how the angle of rotation (holonomy) changes. Such a tool uses simple visual cues (colored arrows, trails) to convey a deep concept (Gauss-Bonnet theorem in this case). In DonutOS, pedagogical overlays might include a **curvature field** visualization – e.g. a grid on a curved manifold panel where moving an object around highlights the net rotation (perhaps by leaving ghost pointers or using an arrow that follows a geodesic). The use of color and trace lines can indicate the "phase" gained around a loop. This not only makes abstract math interactive but provides a **layered overlay** (the user sees the base geometry and a transient layer showing the path and rotation).

Interactive Tensor Manipulation: Beyond static visuals, **manipulable overlays** let technical users experiment. The GuiTeNet example above is one – it allows directly **dragging to contract tensors** [10] or split them, essentially letting the user *play* with the building blocks of a tensor network. Another interactive model is found in quantum circuit editors (like the Quirk circuit simulator) – you drag gates onto wires and the output state updates live. Translating this to DonutOS, one could have an "overlay editing mode" where users compose operations: e.g. gluing one overlay patch to another (more on that in the next section) or connecting output of one analysis panel to input of another. The **compositional overlays** idea suggests that maybe different visual layers (say a Poincaré disk overlay and a Platonic solid overlay) could be *aligned or connected* to illustrate a concept. For example, a user might align a dodecahedron overlay with a Poincaré tiling such that each face of the dodecahedron corresponds to a cell in the hyperbolic tiling – showing a correspondence between a finite polyhedron and an infinite tessellation (this is conceptually related to how some tessellations relate to projections of higher-dimensional polytopes). Technically, interactive alignment could be done by letting users rotate/position overlays relative to each other, with snap-to-fit guides. Visual cues like highlights or ghost images can assist in this process.

Visual Cues for Symmetry and Information Flow: Many demos emphasize *color, animation, and morphing* to convey abstract properties. To illustrate **symmetry**, one might animate a shape morphing into its dual or highlight symmetric pairs of elements. For instance, a Three.js demo could show an **icosahedron morphing into a dodecahedron** (its dual) by interpolating vertex positions – during the morph, corresponding elements could light up in the same color, teaching which face became which vertex, etc. This kind of animated overlay is both engaging and instructive. To show **information flow**, network visualizations often use moving particles or gradients along edges. In a WebGL context, one could have a shader animate a pulsating glow that travels from node A to node B along a link whenever a "contraction" or interaction happens. Using color gradients can indicate intensity or direction (for example, a red-to-blue gradient arrow might indicate flow from a "hot" source to a "cool" sink). **Three.js** can animate object properties every frame, so a custom overlay could, say, depict a wave propagating through a lattice (good for illustrating e.g. a quantum circuit's state propagation or a neural network's signal). These techniques keep the visualization pedagogically clear: *temporal cues* (animation) show *processes*, and *color/size cues* show *magnitudes or categories*. DonutOS can leverage this by, for instance, coloring sections of the torus overlay when a certain mode is active (perhaps tie the color to frequency or phase in a phase-lock overlay). The end result is an interactive model where users **see cause and effect**: when they manipulate a parameter (spin a knob, drag a node, tilt the torus), the holographic/tensor overlay responds immediately with visual changes, reinforcing understanding of the underlying symmetry or curvature.

Comfort and Ergonomics in 3D/VR Overlays

Designing holographic 3D overlays for long usage requires careful attention to **UX comfort**. Key guidelines from VR/AR research can inform DonutOS's overlay system:

- **Manage Visual Complexity and Clutter:** In AR interfaces, it's advised to *"Limit on-screen elements to avoid information overload. Use transparent overlays instead of solid UI components, allowing virtual objects to blend with the real world while maintaining clarity."* [19] This means DonutOS overlays should likely be semi-transparent or minimalist when superimposed on the main view, so they don't overwhelm the user's vision. For example, a Platonic grid overlay might be drawn with faint lines or low opacity fill, just enough to guide attention without blocking other content. Transparency also helps avoid occluding too much of either the virtual or real-world context (important if DonutOS is AR; if purely virtual, it helps stack multiple overlays without each hiding the others completely).

- **Optimal Use of Transparency and Occlusion:** Improper use of transparency in 3D can cause visual artifacts (like flicker or depth sorting issues). It's important to depth-test and order overlay renderings to avoid the "shimmering" or *flickering* that can occur when two semi-transparent surfaces conflict [20]. Three.js offers techniques like using additive blending or alpha-to-coverage to reduce such flicker. Additionally, consider **occlusion handling**: in AR, critical real-world objects (like obstacles or faces) should not be fully occluded by overlays – some systems use **dynamic occlusion** (making the overlay fade out or "window" around important objects). In VR, purely virtual occlusion is less of a safety issue, but one still doesn't want important UI elements hiding each other. A practical approach is to allow users to reposition or dismiss overlays easily to clear their view (perhaps a quick gaze gesture to "peek" past an overlay, or a one-button **reset/clear** to hide all overlays momentarily).

- **Avoid Motion Sickness and Visual Discomfort:** Since DonutOS involves potentially moving, rotating 3D elements (e.g. a rotating torus, orbiting overlays), maintaining comfort is crucial. VR best practices recommend **stable horizons and user-controlled motion**. Sudden or autonomous movements of the camera or UI can be disorienting [21] [22]. DonutOS should ensure any animated overlay motions are subtle and anchored. For instance, if the torus rotates as part of the UI, it might rotate slowly or only in response to user input (no surprise spins). Keep rotation speeds moderate or allow the user to choose the speed. Also, avoid **head-locked flicker**: flashing content in the center of view can induce nausea or even trigger seizures if in the 1–20 Hz range [23] [24]. Thus, any blinking highlight or strobe effect in an overlay should either be above ~60 Hz or kept very mild. Instead of flashing a panel to get attention, use a gentle pulse or a color change.

- **Depth and Distance of UI Elements:** To reduce eye strain, VR guidelines suggest placing UI elements at a comfortable virtual distance (around 0.5 to 2 meters away) and not too close to the user's face [25]. In DonutOS, which features a toroidal "world", the HUD overlays could be drawn on a curved surface at a consistent radius from the center so that the user's focal distance doesn't constantly change. If DonutOS is seen on a flat screen (desktop use), this is less of an issue, but for any stereo or immersive mode it's critical. Also ensure text and icons are sized for legibility at that distance (scaling with user's display). Using **3D depth layering** for overlays (e.g. placing some slightly behind others) can create a natural visual hierarchy without causing confusion, as long as each layer is discernible and not all semi-transparent (which could lead to depth rivalry in vision).

- **Reset and Safety Mechanisms:** Complex 3D interactions can occasionally leave the user disoriented or the UI in an odd state. It's important to have easy **"reset" affordances** – for example, a quick button to re-center the view or realign all overlays to default positions. In VR, apps often let users recenter orientation with a button (setting the forward direction to where they're currently looking) [26]. In DonutOS, a "Reset Overlay" action could default all panels back to their docking positions on the torus, or flatten any rotations. This is especially helpful if an overlay drifted or the user lost track of something. Similarly, provide ways to quickly *toggle off* layers of overlays to reduce cognitive load – e.g. a hotkey to hide all analytical overlays and show a clean view (and toggle back). Ensuring that every state is *reversible* or not permanently confusing aligns with the **weak-control, low-friction** UX principle mentioned in the DonutOS spec [27].

- **Mitigating VR/AR Specific Issues:** If DonutOS is used in AR, consider environment lighting – AR headsets often produce a **dimmed real scene** and overlay bright graphics, which can cause eye strain. It's useful to adopt the XR device standards: for instance, ANSI guidelines for XR recommend ensuring the device's *optical transmittance* is high enough for the given ambient light [28] [29]. While as a software developer you may not control hardware transmittance, you can design overlays that dynamically adjust brightness/contrast based on ambient light (many AR UIs shift to a darker theme in bright sun to remain visible without becoming glaring). Also, field of view is an issue – keep essential overlays within the central 1/3 of the view if possible [30], since peripheral content might be blurry or missed (and on some AR glasses, outside the display field entirely). For motion sickness, **latency** and **frame rate** are crucial: a complex overlay must be rendered efficiently (using Three.js optimization tricks like frustum culling, minimal draw calls, etc.) to maintain a high frame rate and low motion-to-photon latency. Any lag between head movement and overlay update can cause VIMS (visually induced motion sickness) [31].

In summary, adhering to these comfort guidelines will make interactive holographic overlays not only *impressive* but also *usable for extended periods*. DonutOS should aim for a *gentle, user-controlled experience:* overlays that enhance understanding without overwhelming, and always giving the user an easy out (pause, reset, hide) if the experience becomes too intense or confusing. By using transparency smartly, avoiding excessive flicker/motion, and providing ergonomic controls, the system can deliver rich 3D content in a **comfortable, user-friendly manner**.

Category-Theoretic Metaphors and Compositional UI Patterns

DonutOS's vision for "compositional overlays" can draw inspiration from **category theory metaphors** – essentially, treating UI components and data as objects and morphisms that can be composed. In practical terms, this often translates to **node-graph interfaces** or **modular UI building blocks** that snap together. A number of web-based visual programming tools embody this philosophy:

- **Node-Graph Composition:** Tools like **VVVV.js** and **ThreeNodes.js** provide a browser-based node editor where each node is a function/operation and connections represent data flow (composition of functions). In fact, node-graph editors have become ubiquitous in creative coding and game engines because they mirror the abstract concept of composing morphisms: the output of one node feeds into the input of another, analogous to $g \circ f$ in category terms. VVVV.js, for instance, features *"over 300 nodes"* and a web UI to connect them, enabling complex WebGL apps without writing code [32]. It explicitly cites that *visual programming is well-suited for procedural generation* and has made advanced techniques accessible by **blurring the line between design and programming** [33]. This ethos – making composition interactive and visual – is exactly what a category theory-inspired UI would do. DonutOS could include a "composition mode" overlay where different panels or overlays (objects) can be **glued** by drawing connections (morphisms) between them. For example, linking a "Tensor Network overlay" object to a "Geometry overlay" object might create a combined visualization (perhaps mapping a network onto a shape). The interface could enforce rules of composition (only certain types match) similar to how in category theory you can only compose morphisms with compatible domains/codomain. Ensuring that these operations are intuitive (e.g., using drag-and-drop like connecting puzzle pieces) would make advanced compositions accessible to non-programmers, aligning with category theory's high-level abstraction power but in a user-friendly form.

- **Gluing and Patching Metaphors:** In category theory, the idea of *pushouts* or *gluing along an interface* is common (e.g., gluing topological spaces along a shared boundary). Visually, this can be represented by "patching" two surfaces together. We see a literal version of this in geometry: some complex solids are constructed by gluing simpler ones at their faces. The Johnson solids referenced earlier are a great example – many can be formed by joining pyramids or rotunda to other polyhedra [5]. A UI metaphor here could let a user **drag one shape onto the face of another** and snap them, effectively *gluing* them. This is similar to how 3D modeling software allow merging objects, but doing it in an educational overlay context (perhaps showing the resulting shape and highlighting the seam) could illustrate concepts of combination. Beyond geometry, "gluing patches" could refer to connecting different coordinate charts (as in atlas of a manifold). A prototype visualization might show two overlapping maps (say two projections of a globe) with a controllable overlap region; the user could slide them until the features align, demonstrating the idea of an atlas. This aligns with category metaphors by demonstrating how local pieces compose to a global whole.

- **Tiled Overlay Alignment:** A specific interactive idea is aligning tiled overlays – imagine multiple transparent layers, each a tiled pattern (could be a texture or a grid) – and the user can rotate/offset them to see interference patterns or find alignments. This evokes the concept of transformations (rotations, translations) as morphisms acting on objects (the tiled patterns). A concrete example might be overlaying two Penrose tilings and shifting one to see when the pattern coincides or produces Moiré effects. The act of alignment can represent finding a **common structure** (a metaphor for pullback or pushout in category terms, where two structures align on a common sub-structure). The UI could assist by snapping when a certain alignment is exact, and perhaps highlighting the overlapping pattern in a bright color. This not only teaches about symmetry and pattern, but also gives a visceral sense of **matching** (an important concept in category theory, matching interfaces of composable parts).

- **Compositional UI in DonutOS:** The DonutOS spec hints at modular panels ("membranes") and the ability to "compose, dock, and float" them [34]. Pushing this further, one could imagine each panel or overlay exports certain *ports* or *data streams* (like an "intention field" panel might output a value, and a "geometry" panel might take that value to morph the torus). A user could connect these in a wiring overlay, effectively building a **category of UI components**. This is analogous to how in some dataflow programming (LabVIEW, PureData), users connect outputs to inputs. Category theory ensures such compositions are associative and have identity elements – in UI, that could correspond to default passthrough nodes or grouping of sub-compositions. While a typical user need not know the formal math, the system can enforce consistency (no incompatible connections) and offer identity transforms (no-ops) as needed. By exposing the compositional structure visually, *power users* can create new integrated overlays. For example, linking a “spin network overlay” to a “sound synthesis overlay” might let spin states generate ambient sound (a creative cross-modal composition). Each link is effectively a **morphism** mapping an output object to an input object.

- **Metaphors in Practice – PolyHédronisme:** A real-world simple example of compositional geometry is **PolyHédronisme** by A. Levskaya [4]. It lets users generate complex polyhedra by applying Conway operators in sequence (e.g., take a cube, then truncate it, then expand, etc.). Each operation is like a morphism taking one polyhedron to another. Although the interface is text-based (you input a sequence of operators), it embodies *compositional thinking*: small transformations compose to a big result. A visual UI could replace the text with interactive steps – each operator as a block in a chain. The user could have a “polyhedron pipeline” (much like a node graph) – start with a Platonic solid object, then apply “truncate” node, then “dual” node, etc., and see the end result live. If needed, they could reorder or remove nodes, illustrating the (non)commutativity of these operations (order matters in this case, which is a valuable lesson in group theory/categorics). This is essentially a functorial way to explore shape-space. The **compositional UI metaphor** here is that *the user is composing functions (operations) and immediately seeing the composed outcome*, which is exactly the kind of **prototype-friendly, parameterizable** tool the question emphasizes.

In conclusion, category theory metaphors encourage us to design UIs where **complex wholes are built by connecting simpler parts through well-defined interfaces**. By looking at node-based programming, geometric construction toys, and patchwork overlays, we find *implementation-ready patterns* like node graphs, drag-and-snap patching, and functional pipelines. These can be brought into the WebGL/Three.js context of DonutOS to allow designers and technical users alike to **visually script their overlay experiences**. Such a system would not only be extensible (users can create new combinations we developers didn’t hard-code) but also educational – it externalizes the structure of the problem (much as category theory diagrams do) making the UI self-explanatory. By prototyping with these patterns, DonutOS can achieve a uniquely flexible “donut-shaped” operating space where every overlay and control is part of a grand compositional diagram the user can tweak and extend at will.

Sources:

- Exploratory Platonic solids and duals in WebGL [1] [2]; *Polyhedra Folding* demo by P. Kalita [6]; hyperbolic Poincaré disk tiling interactive by M. Christersson [7].
- GuiTeNet tensor network GUI (D3.js) – interactive contraction and code-gen [9] [10]; use of MERA/ tensor networks in browser [35] [8].
- Bloch sphere and parallel transport demos – interactive spin and holonomy visualization [17] [18].
- VR/AR UI comfort guidelines – transparency and overload in AR interfaces [19]; Oculus VR best practices on flicker and motion [23] [24]; reset orientation tip [26]; AR field-of-view and occlusion safety (ANSI standard) [36] [28].
- Node-graph and composition metaphors – VVVV.js visual programming nodes [32] [33]; polyhedron composition via Conway ops in PolyHédronisme [4]; Johnson solids gluing example [5].

---

[1] [2] Platonic Solids - Duals - NLVM
http://nlvm.usu.edu/en/nav/frames_asid_131_g_4_t_3.html?open=instructions

[3] [4] [5] Polyhedra Viewer
https://polyhedra.tessera.li/

[6] GitHub - paaatrick/polyhedra-folding: WebGL demo of folding polyhedra
https://github.com/paaatrick/polyhedra-folding

[7] Non-Euclidean Geometry: Interactive Hyperbolic Tiling in the Poincaré Disc
http://www.malinc.se/noneuclidean/en/poincaretiling.php

[8] [9] [10] [11] [12] [13] [14] [35] GuiTeNet: A Graphical User Interface for Tensor Networks | Journal of Open Research Software
https://openresearchsoftware.metajnl.com/articles/10.5334/jors.304

[15] [16] TensorSpace.js
https://tensorspace.org

[17] IQM Academy - Learn Quantum Computing Online
https://www.iqmacademy.com/play/bloch/

[18] Parallel Transport on a Sphere
https://afana.me/physics/post/parallel-transport-on-sphere

[19] Designing for Augmented Reality (AR) Interfaces: Best Practices and ...
https://www.kaarwan.com/blog/ui-ux-design/best-practices-designing-for-augmented-reality-interfaces?id=1615

[20] [21] [22] [23] [24] [25] [26] [30] s3.amazonaws.com
https://s3.amazonaws.com/arena-attachments/238441/2330603062c2e502c5c2ca40443c2fa4.pdf

[27] [34] DONUT_SPEC.md
file://file_00000000f40871f48312f3e668eb2b63

[28] [29] [31] [36] Frontiers | Visual performance standards for virtual and augmented reality
https://www.frontiersin.org/journals/virtual-reality/articles/10.3389/frvir.2025.1575870/full

[32] [33] VVVV.js - A Visual Programming Framework for High-End Javascript/WebGL Application Development | by 000.graphics | Medium
https://medium.com/@leavingtheplanet/vvvv-js-a-visual-programming-framework-for-javascript-webgl-application-development-b1575bef52d8