ChatGPT

# Compositional Scene Algebra for DonutOS: Category Theory, Topology & Toroidal Attention

## Summary of Key Frameworks and Concepts

| Framework/ Concept | Key References | DonutOS Application |
|---|---|---|
| **Categorical UI Composition** – Scenes as objects; colimits (pushouts, coequalizers) glue components | Pushouts in category theory formalize *gluing* (e.g. identifying common sub-objects) [1] . Colimits like pushouts and coequalizers provide the “most general” way to merge structures [2] . England (2013) pioneered category theory in HCI [3] . Topos Institute’s AlgebraicJulia uses **cospans** (pushouts along interfaces) for diagram composition [4] . | **Overlay Algebra**: Snap/dock UIs by *pushout* gluing along shared boundaries (e.g. a common edge *Z* identified between panels *X* and *Y*) [1] . Merging duplicate elements by coequalizers (identifying equivalent content). Guarantees universal “optimal” composition (no unintended identifications) with predictable behavior. |
| **Sheaves on Cell Complexes** – Data assigned to spatial cells with local consistency | Sheaves “model local phenomena that lead to global effects” [5] . A *cellular sheaf* is a functor $F: \mathsf{CellComplex}\to \mathsf{Sets}$ (or $\mathsf{Vect}$, etc.), assigning data to each cell and consistency maps on overlaps [6] . Joslyn et al. (2014) use sheaves to integrate multi-source info with consistency checks [7] . **PySheaf** library manipulates sheaves on cell complexes [8] . | **UI as Presheaf/Sheaf**: Represent an immersive UI layout (cells in a grid/mesh) as a base space and UI content as a presheaf of data on those cells. Ensures **local agreement** – e.g. an overlay and base UI must agree on shared border state. Sheaf cohomology highlights inconsistent overlaps (no global section) [7] , guiding resolution or tolerant display of ambiguity. |

| Framework/ Concept | Key References | DonutOS Application |
|---|---|---|
| **Topological Data Analysis (TDA)** – Homology of spaces and data distributions | *Persistent homology* and Betti numbers detect holes and connectivity in data [9]. Ghrist et al. use homology for sensor coverage and manifold learning [10] [11]. A Nature 2022 study applied TDA to neural activity, revealing a **toroidal manifold** for grid-cell maps [10]. **Gudhi library** provides tools to compute simplicial complexes and homology [12]. | **DonutOS Analytics**: Model the user's attention or UI interaction graph as a topological space. Use persistent homology to find loops (cyclic workflows, recurring attention patterns) or voids (unexplored UI regions). For example, detect a **toroidal attention field** – analogous to grid-cell attractors on a torus [13] – indicating periodic focus cycles. Identify connected components or holes in AR scene graphs for semantic clustering and anomaly detection (e.g. a Betti-1 hole might signal a closed navigation loop). |
| **Physics & Field Theory** – Holography and gauge invariance on UI "membranes" | The **holographic principle**: all information in a volume can be encoded on its boundary [14]. In AdS/CFT, a 3D bulk corresponds to a 2D boundary field theory [15]. Gauge theories on lattices assign group elements to edges, enforcing invariances (e.g. phase shifts that cancel around loops). The generalized holographic principle asserts that boundaries carry all inter-system information [16], explaining gauge symmetry as no "hidden" bulk state [17]. | **UI Boundary ↔ Bulk**: Treat the DonutOS interface as a **membrane** encoding the state of a 3D immersive scene (bulk). The user sees and interacts only with boundary projections (UI widgets, AR overlays) that fully represent the hidden content – akin to a hologram [14]. **Gauge-invariant UI operations**: changes in internal variables that don't alter the boundary display (like moving a camera frame without changing the rendered image) are considered symmetries. UI overlay alignment can use lattice-like grids with transformations that compensate (similar to gauge fields ensuring a consistent stitching of overlays regardless of coordinate frame shifts). |

| Framework/ Concept | Key References | DonutOS Application |
|---|---|---|
| **Logic & Semantics** – Topos theory contexts, paraconsistent and game semantics | In a *topos*, each context (site) has its own logic; presheaf topoi support varied truth values. **Paraconsistent logic** allows contradictory statements to coexist without explosion [18]. For example, *dialetheic* systems tolerate $P$ and ¬$P$ both true. Game semantics models interaction as a two-player game (User vs System), where interface moves correspond to logical moves [19] [20]. | **Coherent UI Contradictions**: DonutOS can maintain conflicting overlay information (e.g. two AI agents' suggestions) without forcing a single truth – a **paraconsistent UI** that highlights tension but remains usable (no logical collapse) [18]. A topos-theoretic UI might treat each overlay source as a separate *local truth context*, gluing them via functors. **Game semantics for UI**: model user-system interaction as a dialogue game – the UI is the board where the user's actions and system responses are moves. This provides a principled way to handle *partial information* and *turn-based interface logic*, ensuring that the composition of UI elements respects strategies (e.g. a widget expects certain user "moves" to proceed). |
| **Neuroscience & Cognitive Maps** – Grid cells, toroidal attractors, predictive coding | Entorhinal **grid cells** form hexagonal spatial codes; jointly their activity lies on a torus (2D periodic attractor) [10]. Head direction cells map to a circle (1D ring). Continuous attractor networks explain these as low-dimensional manifolds in neural state space [21] [22]. **Predictive coding** theory (Friston) casts perception as minimizing prediction error; **active inference** extends this to action. Recent work uses category theory (Bayesian lenses, open games) to formalize active inference [23]. | **Attention as Toroidal Field**: Model the user's focus as a **toroidal resonance** – e.g. horizontal and vertical attention cycles forming a 2D torus. DonutOS overlays could implement **grid-cell-like coordinate frames**, helping users maintain orientation in AR space. For instance, a HUD with a repeating grid might tap into the brain's spatial periodicity. Predictive overlays: the UI anticipates user needs (next likely action) and presents "holographic" hints – essentially implementing a **predictive coding UI**. By using frameworks like **Bayesian sensorimotor loops** [23], DonutOS can adjust overlays based on learned predictions (e.g. dimming irrelevant parts of a scene, highlighting likely targets), analogous to how the brain's internal model guides attention. |

## Category-Theoretic Models for Spatial UI Composition

DonutOS can be grounded in category theory by treating **spatial structures** (layouts, scenes, or geometric complexes) as objects in a category, and treating compositional operations (snap, overlay, merge) as morphisms. This provides a high-level algebra of UI assembly. In category-theoretic terms, **colimits** are the key: a colimit stitches together a diagram of pieces into a single combined object in an optimal way (universal with respect to that diagram) [2].

- **Objects and Morphisms**: We define objects such as a pane, window, or 3D scene component. A morphism could be an *inclusion* of one UI component into another (e.g. an overlay's attachment mapping). This naturally leads to a *diagram* of UI components. A **pushout** is the colimit of a span $X \leftarrow Z \rightarrow Y$ – intuitively, it glues two objects $X,Y$ along a common part $Z$ [2]. In topology, gluing spaces along a shared subspace is exactly a pushout [1]. In DonutOS, if two UI panels share a boundary region $Z$ (say a matching edge or docking interface), the **snap-to** operation identifies that region in both and unifies the panels. The result is their pushout $P = X \sqcup_Y Z$, essentially a single interface where $Z$ is not duplicated but merged.

*A pushout diagram gluing two UI components $X$ and $Y$ along a shared sub-part $Z$. The resulting composite $P$ (pushout) contains $X$ and $Y$ with $Z$ identified. Morphisms $i_1,i_2$ are the injections of $X,Y$ into the composite* [2] [1].

In the diagram above, think of $Z$ as a "connector" widget or overlapping area that $X$ and $Y$ both include. The pushout $P$ merges $X$ and $Y$ by identifying each point of $Z$ in $X$ with the corresponding point in $Z$ in $Y$ [1]. This formalizes **"docking"**: e.g. attaching a heads-up display overlay onto a live camera view where the coordinate grid $Z$ (perhaps a calibration frame) is the common interface.

- **Colimits for UI Algebra**: Besides pushouts, other colimits like **coproducts** model side-by-side placement (no identification, just a disjoint union of components), and **coequalizers** model factorizing or merging duplicates. For example, if two overlays refer to the *same underlying object*, a coequalizer can quotient out the duplicate representations (identifying two distinct UI elements via an equivalence). This could describe a **merge operation** where two icons representing the same file are unified into one symbol (the equivalence relation "these are the same file" is coequalized). Category theory ensures these compositions are associative and composable themselves – enabling complex UI scenes to be built from small pieces in a principled way.

- **Cospans and Interfaces**: A useful categorical tool in design is the **cospan**, which represents an object with an exposed interface. Imagine each UI widget comes with "ports" or attachment points; a cospan is $A \rightarrow X \leftarrow B$, where $A,B$ are the interface types on either side of component $X$. Composing cospans by pushouts (gluing matching ports) yields larger assemblies [24] [4]. This is already used in system modeling: for instance, StockFlow diagrams (for systems dynamics) are composed via decorated cospans in AlgebraicJulia [4]. DonutOS can adopt the same: define a category of "UI components with ports" and use pushouts to snap them together. The **Overlay Algebra** then becomes a collection of pushout diagrams – a kind of *visual programming language* where the geometry of how components connect has a formal algebraic semantics.

In summary, category theory gives DonutOS a **diagrammatic compositionality**. Every UI scene graph can be seen as a diagram of pieces, and the **colimit** yields the whole. This ensures that whether you assemble the scene all at once or piece by piece, the final result is canonical (up to unique isomorphism) [25] [26]. It also aids **reasoning**: using the universal property, any constraint on the composite UI (e.g. a global style or rule) can be traced to either a constraint on parts or on the glue. This high-level algebra can be implemented with libraries like **Catlab.jl**, which provides colimit computations for user-defined categories (e.g. computing a pushout of two graphs via one function call) [27] [28].

## Sheaves and Presheaves over UI Layouts

To capture data that “lives” on a spatial UI (e.g. content within regions, or values attached to parts of a scene), sheaf theory offers a powerful framework. A **presheaf** on a space associates to every region (open set, cell, etc.) some data, with restriction maps for containment. A **sheaf** adds the condition that compatible local data glues to a unique global data [29]. In DonutOS, we can treat the UI layout or AR world as a *base space* – for instance, a cell complex covering the screen or environment (vertices, edges, faces for GUI elements or 3D zones). A configuration of the UI (text, images, interactive state) is then a *sheaf of data* over that complex.

Formally, let $X$ be the structure (e.g. a planar graph of UI containers, or a 3D mesh of an AR world). One can consider the poset category of cells of $X$ (or open sets in $X$) – for each cell $σ ∈ X$, a sheaf $F$ gives a set $F(σ)$ of data attached, and for each inclusion $τ \subseteq σ$ (say $τ$ is a boundary of $σ$), a restriction map $F(σ)\to F(τ)$ that tells how data on $σ$ looks when viewed on the sub-cell $τ$ [30]. For example, if $σ$ is a rectangular panel and $τ$ is its top edge, a UI sheaf might assign to $σ$ a **CSS style** (colors, etc.) and to $τ$ (the edge) a restricted style (perhaps just the border color). The restriction map forgets most of the panel’s info and just gives the border properties.

The **sheaf condition** requires: if a set of panels ${σ_i}$ covers a larger region $U$, and we have data on each $σ_i$ that agree on all overlaps $σ_i ∩ σ_j$, then there is a unique consistent global data on $U$ that restricts to those pieces [29]. In UI terms, if two adjacent components share a boundary and both assign the same content or parameters on that boundary, we can consistently treat them as part of one larger component. This is crucial for **overlay coherence**: imagine an AR overlay that spans two regions of a headset’s view – each region’s rendering must agree on the overlap (so the image is seamless). The sheaf gluing condition would enforce that the pixel values or features on the overlapping edge match, yielding one continuous image when combined.

- **Local-to-Global and Vice Versa**: Sheaves excel at analyzing when locally defined interface pieces form a coherent global whole. For DonutOS, this can track **consistency of distributed UI elements**. For instance, a heads-up label appears in two different camera views (two regions) – a sheaf can model this by assigning the label text to each view region and requiring equality on the overlap (the overlap might be the 3D location of the object being labeled). If a contradiction occurs (the two views insist on different text for the same object), the sheaf cohomology detects a non-zero 1-cocycle (an inconsistency loop) [7]. This is a cue for the system to resolve ambiguity or alert the user to a conflicting view. In a paraconsistent design (discussed later), the system might deliberately allow a “controlled” inconsistency – effectively maintaining a presheaf (data assignments) that is not a proper sheaf (no single global truth), while still tracking the extent of the discrepancy.

- **Dynamic Overlays and Sections**: A sheaf's global assignment is called a *section*. In interactive terms, a section could represent a particular configuration of all UI elements that is globally valid. **Stalks** $F(x)$ (the data at a point or cell $x$) might represent possible states or content at that location (all possible button states, or all possible AR annotations for that object). A *local section* on a region is like a partial UI state for that part. The act of **merging overlays** is then finding a section on the union of their domains. If none exists (due to inconsistency), the sheaf formalism pinpoints which conditions failed. This could drive an algorithm to adjust one overlay or propagate changes until consistency is achieved – analogous to solving constraint satisfaction via sheaf-theoretic inference [7].

- **Sheaves on Graphs and Cells**: Much of the applied sheaf theory literature deals with sheaves on graphs or cell complexes, which are very relevant here (a UI layout can be abstracted as a graph of connectivity or a 2D cell complex of regions). Hansen's *"gentle introduction to sheaves on graphs"* and Ghrist's work on networks show how to compute sheaf cohomology to find disagreements or gauge degrees of inconsistency [31] [7]. In DonutOS, one could assign a numeric field on each UI element (say an "importance" or saliency value) and use a sheaf to ensure these values locally agree where elements overlap or interface. A discontinuity would indicate a saliency jump – which might be intentional (edge of focus) or something to smooth out via UI transition.

- **Tooling**: The **PySheaf** toolbox provides a programmatic way to define cell complexes and attach data (as sets or vector spaces) with restriction maps [8]. Using PySheaf, a developer could encode a UI's adjacency graph and some data (like user permission levels required for each section), then automatically compute where inconsistencies arise or compute *sections* that extend partial configurations. Likewise, **AlgebraicJulia**'s **CSets** (Combinatorial Sets) can represent similar structured data on graphs, and their colimit operations could combine sheaves as the UI components merge. This allows a flexible, *compositional UI state*: each overlay or module carries its own data (a sheaf on its internal structure), and when we compose modules via category operations, we can also **glue their sheaves** via pushforward/pullback along the attaching maps [32] [33]. The result is a big presheaf or sheaf on the whole scene. The mathematics ensures that if each piece was locally consistent and they agree on interfaces, the whole is consistent – a valuable guarantee for complex, multi-source overlays.

In essence, sheaf theory in DonutOS provides a unifying language for **multi-layered data** on spaces: UI overlay as a section of a sheaf, sensor data mapped onto world geometry as another sheaf, etc., all living on the same base space and combined through colimits of sheaves (which correspond to *limits* in the opposite category, aligning with pullbacks for data integration) [32]. This can bridge the UI's visual layer with the system's underlying data layer in a consistent way.

## Diagrammatic Colimits for Graphical Composition

The snap-together behaviors of a UI can be concisely described with **diagrammatic operations** that are in fact colimits "in action." We highlight some specific ones relevant to DonutOS overlay algebra, and how category theory captures them:

- **Snap/Dock (Pushout)**: As described, snapping an overlay to a base view corresponds to identifying a substructure (e.g. an edge or a coordinate frame). In pure topology, if $X$ and $Y$ are two spaces and $Z$ is a common part (with maps $f:Z\to X$, $g:Z\to Y$), the pushout $P = X \cup_Z Y$ is essentially $X$ and $Y$ glued along $Z$ [34]. The pushout comes with canonical injections $i_1:X\to P$, $i_2:Y\to P$ so that $i_1\circ f = i_2\circ g$ (the two copies of $Z$ coincide in $P$). For UIs, one can think of $Z$ as an **alignment interface** – for example, a bounding box or anchor point. Gluing along $Z$ means the overlay's anchor is identified with the base's anchor point. The result is a unified scene where the overlay is properly attached. The universal property of pushout guarantees that any other way of attaching $X$ and $Y$ along $Z$ factors through this one [25] – in practical terms, it doesn't matter in which order or orientation we perform multiple attachments; the pushout combination is invariant (up to isomorphism), making the snap operation well-behaved in complex scenarios (no dependency on assembly order).

- **Overlay Union (Coproduct)**: If overlays stack or overlap without direct interaction (no common region to identify), the operation is just a **coproduct** (disjoint union). In a category of UI components, a coproduct $X \sqcup Y$ represents placing two elements in parallel without merging any part. This might correspond to drawing one on top of the other with transparency – the structures remain independent. The coproduct is the simplest colimit (just put pieces together, no identifications) [35]. Later, if the user decides these two overlays refer to the same entity, a coequalizer could be applied to merge them.

- **Merge (Coequalizer)**: Consider two morphisms $p,q: X \rightrightarrows Y$ in a diagram, meaning we have two different ways that a subcomponent $X$ is mapped into a larger component $Y$. In UI, this could happen if the same sub-widget appears twice in a composite (two references to one menu, say). A **coequalizer** identifies $p(x)$ with $q(x)$ for all $x \in X$, effectively merging the two appearances into one [36] [37]. In DonutOS, a merge operation could allow the system or user to declare that two overlay elements (perhaps initially separate due to different data sources) are actually the same – the coequalizer then "glues" them and factors out the duplication. The result is a quotient of the disjoint union by the smallest equivalence relating those duplicates [38] [36]. This is analogous to merging layers in a graphics editor: after merge, editing that part affects both former instances, since they've become one.

- **Stack (Monoidal composition)**: Layering an overlay on top of another without geometric identification can be modeled by a *monoidal category* structure (where objects have a tensor product meaning "simultaneous display"). This is not a colimit per se but a separate categorical product. However, one can simulate simple stacking by treating the z-index as an attribute and including it in the presheaf (so two layers don't conflict unless they have the same z-index region). If one wanted a formal pushout model, one could introduce an "empty intersection" for different layers (they meet only in an empty subspace), so the pushout of two layers along an empty interface is just their disjoint union – recovering the idea of stacking without interaction.

These operations compose. For example, docking a window (pushout), then merging some content inside it (coequalizer), then stacking another layer on it (coproduct or monoidal stacking) are all colimit-like operations that can be combined thanks to the universal properties. The **commutativity** of colimit diagrams (under appropriate conditions) means the end result does not depend on the sequence – a huge plus for a user interface that might support drag-and-drop composition and later merging or grouping. This algebraic approach enables *reasoning about UI composition* at a high level: we can ask questions like "is the composition of two snap operations itself a snap of some larger pattern?" – category theory often can answer such questions through pushout-past-pushout lemmas or colimit decompositions [39] (similar to how graph rewriting theories exploit pushout composition).

From an implementation perspective, these ideas can be prototyped with category-theoretic libraries. **Catlab.jl** (AlgebraicJulia) provides a way to define custom categories (e.g. a category of GUI layouts) and then invoke generic colimit algorithms [40] [27]. The outcome of a colimit computation could be a new data structure representing the merged UI. If performance is a concern, specialized methods can be written (the math guarantees existence but you can optimize the actual gluing of data structures). The benefit is that the *correctness* is ensured by the universal property, and one can layer additional constraints (e.g. adhesive category conditions to ensure nice behavior of overlaps). This diagrammatic rigor prepares DonutOS to handle increasingly complex scenes – as AR and VR UIs grow, having a rock-solid algebra of composition will be vital.

## Topological Data Analysis for DonutOS Scenes

DonutOS not only composes scenes – it can also **analyze** them. This is where topology and TDA (Topological Data Analysis) enter. By treating certain structures (or even user interaction patterns) as *shapes*, we can compute their topological invariants to gain insight that is invariant under continuous deformations. Two major tools are **homology** (measuring holes) and **persistent homology** (measuring multi-scale holes).

- **UI Graphs and Nerve Complexes**: A UI or AR scene can be abstracted as a graph or simplicial complex. For example, consider a graph where nodes are UI elements and edges represent adjacency or links (like a navigation graph of interface states). The *homology* of this graph is simple (Betti_0 = number of connected components, Betti_1 = number of independent cycles). A cycle (Betti_1 = 1 or more) in the interface graph might indicate a **loop in user flow** – perhaps the user can cycle through a set of overlays in a way that returns to the start. This could be a design feature (e.g. a circular menu) or a potential confusion if unintended. **Persistent homology** can take a nested family of such relationships (maybe at different interaction frequencies or attention thresholds) and identify which connectivity features persist across scales [41]. This might detect robust clusters of UI states or pathways that remain strongly connected across varying user behaviors.

- **Attention and Gaze Topology**: If we model the user's attention as a point moving in a high-dimensional state space (where dimensions could correspond to focal parameters, like position on screen, depth, context, etc.), we can sample points from this space during use. TDA can form a *point cloud* of these attention states and construct a simplicial complex (via, say, the Vietoris–Rips algorithm at various radii). The **persistent homology barcodes** from this can tell us if the attention states lie on a particular manifold – e.g. a torus or sphere. Notably, the discovery by Gardner et al. (2022) that grid cell ensemble activity lies on a torus was made by applying topological data analysis (computing the persistent Betti numbers) [10]. They found two persistent $H_1$ holes (loops) and one $H_2$ hole, matching a torus's topology. In DonutOS, if we suspect the user's focus cycles through two periodic factors (like horizontal and vertical scanning of a scene) we might confirm this by detecting a toroidal attention manifold (Betti_1 = 2) in the interaction data. A practical outcome: recognizing a torus could mean the UI should support that cyclical scanning with appropriate wrap-around cues (since the user's cognitive map is essentially toroidal [13]).

- **Scene Geometry and AR Cloud**: DonutOS immersive scenes (e.g. a 3D mesh of a room with overlays attached) are inherently spatial. We can apply algebraic topology to the **shapes in the scene**. For example, use homology to detect if the AR annotations around an object form a closed loop (maybe a ring of labels around something). If $H_1$ of the subcomplex of annotations is non-zero, it means there is a "hole" – possibly we've encircled an object with information. This might be used to decide rendering (e.g. if labels encircle an object completely, maybe switch to a different presentation to avoid occlusion). In AR cloud reconstructions, **persistent homology** helps filter noise and find true surfaces – TDA can identify connected components that persist as one grows a radius, corresponding to physical surfaces in point clouds [42]. DonutOS analytics could leverage this by running TDA on spatial point clouds to detect features like tunnels or voids in the environment that might need special UI treatment (e.g. highlighting a doorway which is a topological tunnel in a wall).

- **Persistent Features for UI Adaptation**: Because persistent homology analyzes shapes across multiple scales [41], it can inform multi-scale UI design. For instance, consider multi-scale navigation (zoomable UIs). We can build a filtration of the UI state-space by "zoom level" or detail level. Persistent homology can show which user pathways or clusterings remain through scales. Those that persist indicate *robust interactions* that should be made prominent; those that appear only at fine scales might be secondary. This is analogous to multi-resolution analysis of usage patterns.

One concrete example: **Understanding user journeys**. Represent the set of all interface panels the user visits as a simplicial complex: each single panel is a vertex; each time the user in one session visits two panels in some order, put an edge between them; a triangle if three panels are visited around the same time, etc. This complex can have holes indicating alternate routes. If a big hole persists (meaning users can go A→B→C→A in one path and A→D→C→A in another, forming a loop in the state graph), that might indicate two distinct workflows that converge. The UI might then present a choice upfront (since it's not a linear sequence but a loopy structure). TDA essentially helps **discover the topology of the UX graph**, not just the shortest paths.

The toolkit to do this is accessible: libraries like **Gudhi** (in C++/Python) or **Ripser** (for fast homology) can take adjacency data or point clouds and output Betti numbers and barcodes [12]. For visualization, the **Topology ToolKit (TTK)** can integrate with VTK/ParaView to visually highlight these features [43]. A developer can instrument DonutOS to log user interactions or overlay positions, feed that into a persistent homology pipeline, and get a succinct description (e.g. "two loops and one connected component – likely a torus"). This could even run in real-time for adaptive interfaces (imagine the interface detecting in real-time that the user is oscillating between two views and thus essentially moving on a loop – it could then proactively create a shortcut across the loop's diameter to simplify the cycle).

Topological analysis thus bridges the **quantitative and qualitative**: it gives a high-level, robust description of the shape of UI usage and layout. For a system like DonutOS, which aspires to model attention and context (fractal-holographic field), these are precisely the tools to detect emergent structure. A fractal aspect, for instance, could be investigated by looking at scaling dimensions (fractal dimension estimation is another TDA tool) – if the distribution of attention points has a non-integer dimension (e.g. 1.5), that's a clue of self-similar exploratory behavior. Persistent homology, on the other hand, will tell you the *qualitative* shape (torus, sphere, cluster, tree, etc.). Together, they help tune DonutOS to the *cognitive topology* of its user's interactions.

## Physics and Field Theory Analogies

The Donut of Attention concept — attention as a *toroidal resonance field* — invites analogies with theoretical physics, where fields and symmetries play central roles. We explore how **lattice gauge theory** and **holographic duality** can inform an "overlay algebra" with boundary/bulk interplay and invariances.

- **Holographic Interfaces**: The **holographic principle** from physics states that everything happening in a volume can be described by information on its boundary [14]. In DonutOS, the "volume" might be the rich internal state of an application or environment, and the "boundary" is the UI surface that the user interacts with. By designing DonutOS such that *all relevant state information is projectively represented on the UI overlays*, we ensure a form of holography – the user never needs to reach "inside" the system; the boundary (UI) suffices. This has a parallel in AR: the physical world is the bulk, and the AR overlay is a boundary that carries information about that bulk (e.g. outlines, labels, and guides). A practical example: instead of showing hidden menu states only deep in the system (bulk), provide contextual cues on the periphery of vision (boundary) so the user can infer the hidden state. The **bulk–boundary correspondence** also suggests a method for UI scaling: simple UIs might only encode a subset of the state (like a low-resolution hologram); as the system grows more complex (bulk info increases), the UI might need to encode more (like adding more pixels to a hologram). The principle gives a bound: there's a maximal information density the boundary can convey, related to its area (in physics, entropy ~ area). This could set UI bandwidth limits – e.g., don't try to shove more information than a user's display (or cognitive capacity) can holographically encode at once.

- **Gauge Symmetry in UI**: Gauge theories deal with fields that have certain transformations leaving physics invariant (e.g. adding a gradient to a potential field doesn't change observable forces). In UI terms, consider a **reference frame change** – say the user moves their head in AR: the internal coordinates of objects all shift, but the relative overlays remain put. This is analogous to a gauge transformation (change of coordinates) that should not affect the gauge-invariant quantities (the relations among objects). We can introduce the idea of a **gauge group** for UI alignment. For instance, a 2D overlay might have a "phase" corresponding to an angle; if two overlays are linked, rotating the underlying coordinate by some angle and simultaneously rotating one overlay's internal alignment by the same angle leaves the overall display alignment invariant. The system might treat this as a $U(1)$ gauge symmetry (like a phase shift). Ensuring **gauge invariance** means the user experience is consistent and doesn't depend on arbitrary choices like coordinate origins or units. If two modules in DonutOS exchange data (say, one overlay uses coordinates provided by another), adopting a gauge-theoretic approach would have the data exchange include transformation rules such that if one module changes its frame, the other compensates. Technically, one could assign transformation group elements to connections between overlays, similar to lattice gauge theory where each link of a lattice has a group element (ensuring a notion of parallel transport). If the user moves in a loop and returns to the same spot (like a closed path in state space), the product of group transformations around that loop could indicate a **holonomy** – if it's not identity, the system knows a context shift happened (e.g. the user's orientation changed after a full cycle, possibly indicating a 360° turn – a $2π$ phase). Designing UI operations to account for this (maybe by gradually adjusting orientation of content to cancel the holonomy) can prevent disorientation.

- **Toroidal Fields and Resonance**: The torus as a shape appears both in cognitive science (grid cells) and physics (toroidal magnet fields, ring lasers). If attention is a toroidal field, one can imagine

**resonant modes** on the torus – analogous to standing waves. For a donut-shaped attention field, there might be natural “frequencies” or patterns the user cycles through (e.g. a horizontal scan vs a vertical scan around the torus). In field theory, modes on a torus are quantized by two angular quantum numbers. In DonutOS, we could use this by analyzing user behavior in Fourier modes: does the user have a dominant horizontal oscillation (one quantum in one direction) combined with, say, two oscillations vertically? That might correspond to an attention mode (perhaps scanning 2 times up-down for every 1 time left-right). Identifying such a mode could allow the UI to **resonate** with the user – maybe by subtly guiding the eyes in that pattern (e.g. highlighting items in sync with the user’s inherent scanning frequency).

- **Boundary Conditions and Membranes**: In gauge theories on a lattice with a boundary, one often sets boundary conditions (e.g. fixed or periodic). DonutOS can analogously allow *periodic boundaries* for UI if it exploits the torus: e.g., an infinite scrolling list is topologically a cylinder; if it’s made cyclic (looping end-to-end), it becomes a torus. This can be beneficial for continuous browsing (the user can scroll indefinitely and return to start seamlessly – an identified torus). The system might detect when a list is conceptually circular (like menu options that conceptually wrap around) and enforce that as a boundary condition – giving a more coherent user experience (no abrupt stops, preserving momentum – a kind of momentum conservation akin to physics).

- **Holographic Analytics**: There is also a feedback in the other direction: just as gauge invariance can be derived from imposing that everything is encoded on boundaries (no hidden channels) [16] [17], we can *use* gauge invariants to monitor system health. For instance, define some quantity of the UI that should be invariant under user viewpoint (like the number of visible markers, which should remain constant no matter the angle, if designed so). If the user’s view change (a big gauge transform) suddenly changes that quantity, it signals a breach of gauge invariance – maybe an overlay isn’t sticking correctly (it disappears at certain angles, etc.). This is analogous to a gauge anomaly. By designing DonutOS with explicit invariants (perhaps using cohomological checks, as with sheaves, to ensure no inconsistencies on overlaps), the system maintains a kind of *conservation law* for UI information.

In summary, physics analogies enrich DonutOS in two ways: **conceptually**, by inspiring features like “everything important on the boundary” (holography) and “no absolute frame for overlays” (gauge symmetry), and **analytically**, by providing tools like lattice discretization of transformations and mode analysis of attention fields. The result is a UI architecture that is robust under change of perspective (literal and figurative) and that echoes deep principles known to govern stable, self-consistent systems in nature.

## Logical and Semantic Enrichments

Interfaces are not just geometric; they carry meaning and sometimes ambiguity. DonutOS can benefit from advanced logic frameworks to handle scenarios where information is incomplete, context-dependent, or even contradictory. **Topos theory**, **paraconsistent logic**, and **game semantics** offer routes to making the UI *semantically aware* and *resilient to inconsistency*.

- **Topos Theory and Contextual Logic**: A topos is essentially a universe of sets with its own internal logic (often intuitionistic). In UI terms, each *context* (say, a mode of the application or a user persona) might be seen as its own topos, with certain truths. Using a topos-theoretic approach, DonutOS could maintain multiple coexisting logical environments for different overlays. For example, an AR overlay from an expert system might use a certain ontology (in which a statement $P$ is true), while another overlay from a social feed might not recognize $P$ at all or consider a related $P'$ true. Instead of forcing one global boolean, we keep their contexts separate (two topoi), each internally consistent. The composition (gluing of contexts) then might happen in a presheaf topos (the category of functors from context index category to $\mathsf{Sets}$), which can accommodate the differing views. Essentially, DonutOS can act like a **stack of logics**: the core OS ensures they communicate without violating each one's internal rules. Topos theory provides the mathematical foundation to do this systematically, as it formalizes how local logic can glue into a pseudo-global logic (sometimes requiring sheaf conditions to hold for overlap of contexts).

- **Paraconsistent UI**: User interfaces often display information that might be wrong or conflicting (think of two apps giving different estimates for the same metric). A traditional system might flag this as an error or force consistency by choosing one. A **paraconsistent approach** instead allows contradictions to be *present but controlled*. Paraconsistent logic is *inconsistency-tolerant*: $P$ and ¬$P$ can both be present without the system exploding into nonsense [18]. In DonutOS, this could mean an overlay algebra that supports **overlays in opposition** – e.g., a "devil's advocate" overlay that deliberately contradicts the main overlay to spur critical thinking. Technically, we can implement this by not using a classical logical layer to aggregate overlay content, but a paraconsistent one (such as a **dialetheic logic** where some propositions are both true and false). The UI would then need to convey the contradiction visually (perhaps highlighting inconsistencies). The important part is that the presence of a contradiction does *not* cascade into every statement being considered true (logical explosion). For example, two AR annotations might label an object as different things – rather than the system breaking, it could show both labels with a warning symbol indicating contradiction, but other unrelated overlays remain unaffected. This way, DonutOS can mirror the real world's complexity (where conflicting information exists) and assist the user in navigating it, rather than hiding or arbitrarily resolving the conflict.

- **Coherent Overlap and Many-Valued Logic**: Another logical angle is using *many-valued* or *fuzzy logic* for overlay blending. If two overlays partially contradict, instead of a binary conflict, one could assign truth values in [0,1] to certain statements (like confidence levels from different sources) and display a combined view that reflects uncertainty (transparency or fuzzy boundaries on the overlay). Topos of *sheaves of truth values* (heyting algebras) could underpin this, ensuring any inference made on the combined info is sound under uncertainty. This ties into *possibilistic UI* design – the interface might show multiple possible states (like ghosted UI elements that may or may not become relevant) without committing to one, thus communicating uncertainty.

- **Game Semantics for Interaction**: Every UI can be seen as a language where the user and system have a dialogue (the user issues commands or gestures, the system responds). **Game semantics** formalizes this by treating the interaction as a game: the system (Proponent) and user (Opponent) alternate moves, and the "meaning" of an interface element is given by how it enables certain sequences of moves. For instance, a dialog box enforces a certain game (user must click OK or Cancel – a binary move). Composing UI elements corresponds to combining games (often in concurrent or sequential ways). Using game semantics, one can reason about *interface equivalence*: is two clicks in menu A followed by one in B equivalent to one click in B then two in A? If the underlying game (strategy) is the same, then the UI flows are equivalent. This could feed into optimization: DonutOS might detect when the user is effectively playing the same "game" through a convoluted sequence and suggest a shortcut that provides an equivalent outcome with fewer moves (a winning strategy). Moreover, game semantics can model *partial information*: one player might have a hidden state (the system has internal info the user doesn't know). The UI is the medium that reveals bits of that hidden state (like hints). By analyzing the UI as a game, we can adjust how much info to present at once to keep it challenging enough but not impossible – akin to balancing a puzzle's difficulty by how many clues (moves) the system provides.

- **Semantic Web and Ontologies**: DonutOS overlays could be backed by semantic data (ontologies describing the objects in view). Category theory comes into play with **institution theory** or **ontological category theory**, where different ontologies are related by functors (interpretation mappings). If DonutOS pulls data from different knowledge graphs, we can use categorical **lenses** or **ontological pushouts** to merge vocabularies. Essentially, each overlay might carry its own semantic context (its own category of concepts), and overlay algebra needs to align these. A pushout in the category of ontologies (via a common subontology) could unify terms. This is high-level, but practically could mean if one overlay calls a concept "Car" and another calls it "Automobile", and an alignment $Z$ says Car=Automobile, the pushout ontology will treat them as one concept and thus merge the overlays' info on that item. Tools like **category-of-ontologies with mappings** are an active research area; DonutOS could leverage them for a seamless semantic user experience where multiple data sources overlay without confusion.

- **Coherence and Exactness**: In topos theory, the concept of *colimit preserving functors* or *exactness* can ensure that logical constraints are preserved when moving between contexts. For DonutOS, we want that if each overlay individually satisfies some logical rules (no internal contradictions, or certain safety rules), then when composed, these rules still hold – or if they fail, we can pinpoint which colimit (merge) caused it. This is analogous to ensuring the UI doesn't inadvertently create a new problem by combining apps (like two safety warnings canceling each other out visually). By modeling the UI composition as functors between logical systems, one can enforce that certain subcategory (of safe states) is stable under colimits.

The bottom line is that an overlay algebra augmented with modern logic can handle **ambiguity and inconsistency** in a principled way. Rather than the ad-hoc solutions of today (where UIs often just avoid showing conflicting info, or pop up raw error messages), DonutOS could present a *coherent contradiction* – clearly visualized and managed – allowing the user to engage with it. It's a very human-centric approach: humans are used to dealing with conflicting information, and providing tools (like comparing overlays side by side, highlighting differences) can turn a potential confusion into a powerful analytical feature. By using frameworks like paraconsistent logic [18] in the backend, we ensure the system itself remains stable and doesn't propagate errors. And by using game semantics and topos theory, we ensure the user *semantics* of the interface – what does this combination of overlays mean? – is well-defined and traceable to the individual parts.

## Neuroscience and Predictive Coding Inspirations

DonutOS is envisioned as an "Attentional OS," so it makes sense to draw from cognitive science and neuroscience for its design. Concepts like **grid cells**, **toroidal attractors**, **predictive coding**, and **active inference** can directly inform how overlays behave and adapt.

- **Cognitive Maps and Grid Cells**: The discovery that the brain's navigation system (entorhinal cortex) contains grid cells whose joint activity maps onto a torus [10] is highly relevant. It suggests the brain uses elegant toroidal coordinates to represent 2D space. If DonutOS models user attention in a spatial interface, it could attempt a similar internal map. For instance, the system could maintain two phase variables $(\theta_x, \theta_y)$ that track the user's sweeping of attention horizontally and vertically across an AR scene. As the user scans repetitively, these angles increase modulo $2\pi$. A complete scan around brings the phase back – hence these coordinates live on a torus (like angle-angle). The *advantage* of this is that periodic behavior is naturally accounted for. Regular patterns in user scanning would correspond to straight lines on the torus (since on a torus, constant movement in one direction eventually wraps around – potentially matching periodic revisits of a spot). DonutOS might even implement artificial "grid cells" to predict where the user will look next. Grid cells in the brain fire periodically as the animal moves; analogously, DonutOS could have a set of virtual sensors (grid overlays) that peak when the user's gaze enters certain zones in the field of view, tiling the space. This could be used for efficient indexing of content: instead of continuously tracking exact gaze, the system knows when the user is near a grid hotspot and can proactively load information for nearby hotspots (like how grid cells can enable path integration). If the user's attention state indeed forms a toroidal attractor, the UI might get into a *resonance* with it: for example, subtle oscillations in UI element emphasis could be timed to the intrinsic toroidal cycle of attention, to guide the user smoothly (this is speculative, but one could imagine a UI rhythm that aligns with the user's scan rhythm).

- **Fractal Attention and Multi-scale**: The "fractal–holographic" phrasing hints that attention might have self-similar patterns at different scales (fractal) and that each part of the attentional field might contain a miniature of the whole (holographic). In practical terms, this could mean the user's pattern of focusing on big areas vs. small details might repeat. Neuroscience shows 1/f scaling in many cognitive signals (suggesting fractal temporal dynamics). DonutOS could monitor the frequency spectrum of the user's interaction intervals; a heavy 1/f component implies no dominant timescale (fractal-like behavior). To accommodate this, the UI could ensure that information is available at multiple scales: e.g., high-level summary overlays for broad glances, and detail overlays that recursively reveal finer content when zoomed. A *fractal interface* might literally use similar patterns at different levels – for instance, a mini-map overview at the corner that is a smaller copy of the main view, encoding global state (hologram-like). When the user interacts with that mini-map, it affects the main (since it's the same info, just scaled). This resonates with the idea of *presheaves on a basis* – each piece of the UI might mimic the structure of the whole UI in some way, enabling users to shift focus without losing context.

- **Predictive Coding and Active Inference**: Predictive coding theory says the brain constantly predicts sensory input and only errors (differences) propagate upward. Interfaces can apply this by predicting what the user will do or need next, and preparing that overlay in advance. For example, if the system's model predicts the user will click a certain button next, it could highlight it or even enlarge its hitbox slightly (just as the brain biases perception toward expected stimuli). If the user deviates (prediction error), the interface can quickly adjust (maybe highlight something else). **Active inference** goes further: not only predicting sensations, but taking actions to fulfill predictions (i.e., make the world match the prediction). In a UI context, this could mean if the system predicts the user is trying to achieve X, it will proactively take steps toward X (like auto-filling a form or guiding the user). However, to avoid annoyance, this needs to be done in a transparent, user-controlled way. Category theory has been used to formalize active inference policies as compositional Bayesian "open games" [23] . DonutOS could incorporate a library of such *active inference widgets*. For example, a navigation overlay in an app could internally run an active inference model: it has a prior that "user wants to go to a frequently visited location", then it sees evidence (user opens map, lingers around a familiar area), and it then *acts* by highlighting that location or even saying "press here to navigate home." If the user indeed wanted that, the prediction is confirmed and the task is sped up. If not, the user can ignore it, and the model updates (prediction error leads it to adjust its belief).

- **Neural Network Integration**: The attention field and UI usage can also be fed into deep learning models (e.g., LSTMs or transformers) that learn the user's patterns. What's important is that DonutOS might treat these as approximating a *manifold* of user states. Modern techniques like **manifold learning** could be applied to high-dimensional interaction logs to find a low-dimensional structure (maybe akin to how grid-cell torus was discovered via dimensionality reduction and TDA [44]). If such a structure is found (be it toroidal or something like a Möbius strip of context switches), DonutOS could then reconfigure its UI to match that structure. For instance, if it finds the user's tasks lie on a cycle (morning routine apps → work apps → evening apps → back to morning, making a loop), it could present the app switcher as a cycle (a ring menu) instead of a random grid, to facilitate moving along that cycle.

- **Memory and Holographic Brain**: There are theories (Pribram's holonomic brain theory) that memory is holographic. If DonutOS wanted to mimic a *holographic memory*, it might store user interaction histories in a distributed fashion across the UI context, rather than a central log. For example, each overlay could keep a summary of how it was used (like a small "engram"). When overlays combine, their summaries overlap, producing an interference pattern that could help recall combined usage scenarios. This is very speculative, but imagine a search function that doesn't just search a log database, but reconstructs likely past scenes by superimposing pieces of recalled overlay states – akin to how a hologram piece can reconstruct the whole image albeit at lower resolution. The fractal aspect suggests doing this at different time scales (recent usage vs long-term usage patterns).

In concrete terms, DonutOS can use insights from neuroscience in the following way: **design UI paradigms that align with brain paradigms.** Toroidal coordinate for spatial interaction is one; **head-direction cells** inspire using a compass-like overlay that is always in the user's peripheral vision to indicate orientation (because head-direction is a 1D circular manifold [21] – a simple compass satisfies that representation). **Place cells** (linked to specific contexts) suggest the OS could mark certain UI states as "places" and have a quick recall overlay (like mental bookmarks) for them when the user is in their vicinity in state-space.

Additionally, **multi-sensory integration** in the brain often uses predictive cues (like we lip-read to predict sounds). DonutOS could use one modality to prepare another – e.g., if the gaze (vision) overlay indicates the user is focusing on an object, the haptic feedback could subtly confirm by a slight pulse – tying modalities together to strengthen the user's brain's predictions (this crosses into UX design beyond pure logic, but it's about speaking the brain's language of expectation and reward).

The Topos Institute example of **Bayesian lenses and statistical games for active inference** [23] shows that we can rigorously compose these cognitive models. A complex attention model could be built from simpler ones (one for spatial focus, one for task intention, one for fatigue level, etc.), each a lens that updates beliefs and actions. The overlays could be driven by these sub-models (like one overlay dims when the "fatigue" model says the user is tired). The key is compositionality – to avoid a monolithic brain model and instead have modular ones that plug into corresponding UI modules.

In short, neuroscience can inspire **UI metaphors that are brain-friendly** (toroidal maps, grid-like layouts, periodic cues) and **adaptive algorithms that align with cognitive processes** (predict, then confirm via feedback). DonutOS sits at the junction of user and machine; by incorporating the user's cognitive architecture into its design, it effectively becomes an extension of the user's mind – the ultimate goal of any interface.

## Tools, Libraries, and Prototypes Aligning with this Vision

Bringing all these theoretical ideas to practice is non-trivial, but there are emerging tools and research prototypes that demonstrate facets of the DonutOS vision:

- **AlgebraicJulia & Catlab**: This suite (developed with the involvement of the Topos Institute) provides a concrete way to encode and manipulate category-theoretic structures in code [45]. Using **Catlab.jl**, one can define a category of UI components and compute pushouts, pullbacks, etc., effectively letting the computer handle the "glue math." AlgebraicJulia's **Semagrams.jl** is particularly interesting – it's a tool for **semantic diagrams** where the diagram's visual form is tied to machine-readable meaning [46]. A Semagram could be, for instance, a flowchart UI that the machine understands as a program. This hints at a future where drawing an overlay isn't just visual – it actually constructs a presheaf or graph that the OS can reason about. DonutOS could use a semagram-like approach for user-created mashups: a user graphically overlays a timeline on a map; because it's semantic, the OS knows it means "animate this route over time."

- **SheafSystem and PySheaf**: **SheafSystem.org** and the `pysheaf` library [8] are efforts to provide computational support for sheaves, cohomology, consistency radius calculation, etc. A developer could, for example, represent the network of AR sensors and their coverage areas as a sheaf and use PySheaf to compute if all sensor inputs can be globally consistent (as Ghrist and others have done for sensor fusion) [31]. In a UI, if multiple sensors or modules provide overlapping information on an overlay, PySheaf's consistency algorithms (e.g. computing the *consistency radius* [47]) could quantify how well they agree. There's even a method `Sheaf.FuseAssignment()` that tries to adjust values to improve consistency [48] – think of it as an automated way to reconcile conflicting overlay data by tweaking them minimally (like adjusting two maps to align). Incorporating such libraries allows DonutOS to perform **live consistency checks** and possibly automated resolution across overlays.

- **Topology ToolKit (TTK)**: An open-source library that integrates with visualization tools to perform TDA (like extracting topological features from scalar fields, computing persistence) [43]. If DonutOS has a performance analytics mode, TTK could be used to examine user interaction heatmaps or attention fields as scalar fields, identifying features (maxima where user spends most time, loops of frequent transitions, etc.). The output could then feed back into UI design (maybe automatically suggesting UI layout improvements by identifying "bottleneck" regions where all interaction pathways converge).

- **Machine Learning Frameworks**: Libraries like **TensorFlow** or **PyTorch** could be utilized to implement predictive coding models (as deep predictive models) or to mimic grid cell representations (some AI research already tries to induce grid-cell-like representations in agents). There's also **Spaun (Nengo)**, a cognitive architecture modeling tool that could simulate aspects of user cognition. While not a library for UI per se, it underscores that cognitive modeling is within reach. DonutOS might have a plugin where a simplified *neural model of the user* runs in parallel, constantly updating predictions of what the user will do next, which the OS uses to schedule tasks. This is speculative, but ties into the *active inference* concept where the system has a generative model of the user.

- **AR Toolkits and Game Engines with ECS**: Engines like Unity or Unreal use Entity-Component-System (ECS) architecture, which is actually amenable to categorical treatment (an entity with components is like a sheaf: it assigns a value in each component "space"). Unity's newer DOTS/ECS could potentially be seen as a category of entities with certain monoidal operations for combining them. Moreover, these engines support *snap-dock* in their editors, but one could imagine a higher-level API where, say, one just specifies "glue this overlay to that along boundary" and the engine handles the transform – effectively exposing the pushout operation. Some research prototypes (e.g., tools that allow visual programming of AR behavior by connecting nodes) align with this idea of a *graphical algebra*. **Unreal's Blueprint** system, for example, is a graph-based programming language – a cousin to the idea of composing via diagrams. The gap is that these are not yet based on formal category semantics, but efforts like **Enso** (formerly Luna, a visual functional programming language) show movement toward more rigorous semantics in graphical form.

- **Knowledge Representation and Ontologies**: For semantic integration, libraries like **OWL API**, or **category-theory-based** ones like the Knowledge Graph toolkit, could help. There's research on functorial data migration (FQL / CQL by David Spivak et al.), where one can formally map data between schemas using category theory. In DonutOS, if two overlays come from different data schemas, a categorical query language could translate one to the other on the fly via a pushout in the category of schemas. This is cutting-edge, but tools exist in academic form.

- **Paraconsistent Reasoners**: Not many mainstream, but there are theorem provers and databases (like some implementations of **Prolog** or custom logic engines) that allow inconsistent facts without trivialization. For demonstration, one could hook up a **Minisat (SAT solver)** that is modified for paraconsistent logic to a UI: for instance, an overlay could run a background check "is everything consistent?" via a solver. If not, instead of failing, it returns which clauses conflict. The UI could then highlight those conflicting facts in red. This is a rudimentary way to handle inconsistency which could be improved with true paraconsistent logic programming languages (research prototypes exist).

- **Human-in-the-loop Tools**: The game semantics viewpoint resonates with tools for usability testing (like stateflow analyzers). One could conceive a tool that automatically extracts the *interaction game* from a prototype UI by exploring it (like a model checker) and then uses game semantics to verify certain properties (e.g., user can always reach a cancel move – a form of determinacy or at least reachability in the game). Academic tools in HCI for model-based UI evaluation (such as task modeling with Petri nets or game graphs) could be extended using category theory to compose models of sub-UIs. Though not off-the-shelf, it's an area where DonutOS could break ground by integrating formal verification into UI design.

In sum, while DonutOS's vision is ambitiously interdisciplinary, there are glimmers of it in various domains' tooling. By combining **AlgebraicJulia's compositional frameworks** [49] [39] for building the structure, **sheaf libraries** [8] for ensuring consistency over that structure, **TDA tools** for analyzing emergent patterns, and **cognitive modeling** for user-adaptive behavior, one can start constructing a prototype that embodies the *overlay algebra*. Each piece addresses a slice: category theory for composition, topology for analysis, physics for analogy-driven design constraints, logic for handling information content, and neuroscience for aligning with user's mental patterns.

The ultimate promise of DonutOS is to create an interface that is **compositional, context-aware, and cognitively resonant**. It means an OS where you can fluidly snap together new experiences (like Lego blocks, but backed by math ensuring they fit), where the system can detect higher-level patterns (a loop of actions, a conflicting piece of info) and adjust or inform the user, and where interacting with the digital world feels as natural as our brain's own navigation and thinking processes. The research reviewed here provides a foundation – an "overlay algebra" grounded in category theory and enriched by topology, semantics, and neural principles – to make such an OS possible.

---

1 2 25 26 34 35 36 37 38 Pushout (category theory) - Wikipedia
https://en.wikipedia.org/wiki/Pushout_(category_theory)

3 Retracting a space onto a piece | Download Scientific Diagram
https://www.researchgate.net/figure/Retracting-a-space-onto-a-piece_fig3_267239476

4 23 24 39 45 46 49 Collaborative modelling – Topos Institute
https://topos.institute/work/collaborative-modelling/

5 31 ntrs.nasa.gov
https://ntrs.nasa.gov/api/citations/20220015406/downloads/WiSEE_STINT_2022_Sheaf_Theoretic_Networking_Status_221013.pdf

6 7 9 29 30 Microsoft Word - TDO Revised (1).doc
http://ceur-ws.org/Vol-1304/STIDS2014_P2_JoslynEtAl.pdf

8 47 48 GitHub - kb1dds/pysheaf: Python Cellular Sheaf Library
https://github.com/kb1dds/pysheaf

10 11 13 21 22 42 44 Toroidal topology of population activity in grid cells | Nature
https://www.nature.com/articles/s41586-021-04268-7?error=cookies_not_supported&code=94c91d56-8b33-45a9-97f2-ae144b6d0ddf

12 The Gudhi Library: Simplicial Complexes and Persistent Homology
https://link.springer.com/chapter/10.1007/978-3-662-44199-2_28

14 15 HOLOGRAPHIC PRINCIPLE. What if our universe is just a... | by Astrocet | Medium
https://medium.com/@astrocet2020/holographic-principle-8b7cec00e5a5

16 17 sites.socsci.uci.edu
https://sites.socsci.uci.edu/~ddhoff/why-holography-fp_2_June_2018

18 Paraconsistent logic - Wikipedia
https://en.wikipedia.org/wiki/Paraconsistent_logic

19 Game semantics - Wikipedia
https://en.wikipedia.org/wiki/Game_semantics

20 [PDF] Game Semantics - Computer Science
https://www.cs.uoregon.edu/research/summerschool/summer18/lectures/ghica1.pdf

27 28 40 Categorical algebra · Catlab.jl
https://algebraicjulia.github.io/Catlab.jl/v0.9/apis/categorical_algebra/

[32] [33] Laplacians of Cellular Sheaves
https://www.jakobhansen.org/publications/thesis.pdf

[41] Topological Data Analysis with Persistent Homology - Medium
https://medium.com/@deltorobarba/quantum-topological-data-analysis-the-most-powerful-quantum-machine-learning-algorithm-part-1-c6d055f2a4de

[43] Persistent Homology for Dummies - the Topology ToolKit
https://topology-tool-kit.github.io/persistentHomologyDummies.html