ChatGPT

Category Theory and the DonutOS: *Foundations and Futures of a Compositional "Attention-Torus" UI*

Introduction: A Toroidal Vision Meets Mathematical Rigor

The **DonutOS** – "Donut of Attention" – is an experimental interaction layer that renders human attention as a **toroidal, fractal–holographic field across scales** [1]. In simpler terms, DonutOS envisions the user's focus as a dynamic donut-shaped space (a torus) that can encompass personal, planetary, and even cosmic scopes. This imaginative paradigm (the "attention-torus") is not only a visual metaphor; it suggests a deeply **composable** user interface (UI) where every element, from floating panels to immersive scenes, can connect and interact in a holistic field. Achieving this vision, however, requires more than creativity – it demands a unifying formal foundation to manage complexity. **Category theory**, a branch of mathematics centered on abstract relationships and composition, offers exactly that foundation. It provides a rigorous language of **objects** and **morphisms** (arrows) to describe components and their interactions, and it emphasizes composition as a first-class principle. This paper explores how category theory can illuminate DonutOS's design and guide its future, blending mathematical rigor with the project's speculative, even mythic, imagination.

Modern UI development often struggles with the lack of a principled formal model. As noted by researchers, one major cause of GUI complexity "may be the lack of a unified formal basis or abstract model for user interfaces... GUI libraries are often described in informal terms and lack constructs for combination and composition of interface components," relying instead on ad-hoc imperative code [2]. Category theory directly addresses this gap by formalizing *composition* – the assembly of simple components into complex wholes – as an algebraic principle. In software, many design patterns implicitly embrace category-theoretic concepts. For example, the *Unix philosophy* of composable tools and the use of pipelines mirror the idea of arrows (data transformations) composing sequentially [3]. Functional programming patterns make this connection explicit: **functors** and **monads** in code are direct translations of category-theoretic functors and monoids [3]. Every time a developer maps a function over a collection, they invoke the **Functor** concept (applying an operation uniformly to all elements) – a structure-preserving mapping between categories [3]. Every time asynchronous operations are *chained* (e.g. promises or futures in UI events), they are using **monadic composition**, which ensures that the sequence of actions behaves as a single unit [3]. These abstractions, borrowed from category theory, demonstrate the power of compositional thinking: by obeying algebraic laws, software components can be combined freely without unexpected side-effects.

Category theory in UI design is thus about discovering the right abstractions so that building a complex interface is like building with LEGO pieces – pieces fit together through well-defined interfaces (morphisms) and obey rules of composition (functorial mappings and algebraic laws). DonutOS, with its highly modular architecture of *membranes*, *overlays*, and *scenes*, is an ideal candidate for a categorical approach. Each UI element can be viewed as an **object** in a category, and interactions or data flows between elements as **morphisms**. The entire state of the UI can be treated as a mathematical structure that evolves through composed transformations. The benefit of such a perspective is twofold: **(1)** it guarantees *consistency* – when designed as functors or natural transformations, changes in one part of the system reliably

propagate to others without breaking invariant relationships; **(2)** it enables *extensibility* – new features can be "glued on" via categorical constructions (e.g. pushouts, discussed later) without rewriting the whole system, much as new mathematical objects can be constructed from old ones using universal properties.

This paper is structured as a science-style exploration that first introduces the role of category theory in UI software design (grounding abstract concepts in current, validated UI paradigms), then delves into an analysis of DonutOS's existing design through a categorical lens, and finally launches into speculative future directions for DonutOS where category theory suggests powerful new UI capabilities. We will see that *composable user interfaces*, *reactive programming*, *scene graphs*, the classic *Model-View-Controller (MVC)* architecture, and layered *overlays* all have natural interpretations in category-theoretic terms such as functors, presheaves, and monoidal categories. Building on these insights, we will articulate a possible **"Scene Algebra"** for DonutOS, envision **node-graph overlays** for visual programming, describe how **pushout** constructions could "glue" interface modules safely, and outline how ensuring **state consistency** becomes an exercise in maintaining functorial relationships. We also speculate on integrating DonutOS's mythic metaphors – from **Platonic solids** to **levogyre** (left-spiraling) **shells** – into this formal framework, showing how imagination and mathematics can co-inform the design. Finally, we propose concrete implementation strategies: building **node editors** for end-user programming, supporting **grouping/ungrouping** of overlays as algebraic operations, mapping between **3D and 2D views via functors**, and using algebraic consistency checks as a form of **safety design** (preventing inconsistent or dangerous UI states).

By grounding DonutOS's fanciful vision in category theory, we aim to demonstrate a path toward a UI that is at once **rigorously correct** and **wildly creative**. The *attention-torus* may be a poetic image – evoking, perhaps, the ancient Ouroboros or a cosmic halo of awareness – but underneath, we imagine a solid backbone of commutative diagrams and functorial mappings that ensure every interaction is consistent and meaningful. In the sections that follow, we journey through this blend of the abstract and the concrete: from the mathematical essence of today's UI patterns to tomorrow's toroidal interface algebra.

Category Theory in UI Design: Compositional Foundations

Category theory provides a unifying language to describe systems in terms of **objects** (abstract entities, like UI components or data states) and **morphisms** (arrows representing relationships or transformations between objects). Two fundamental axioms govern categories: **identity** (every object has a do-nothing arrow to itself) and **associative composition** (arrows can be composed end-to-end, and this composition is associative in order) [4] [5]. These simple rules yield profound consequences. In a UI context, an **object** might be a particular screen state or a UI element (e.g. a panel, a button, or an overlay), and a **morphism** could be an interaction or data flow (e.g. "clicking button X triggers action Y", or "the output of panel A is fed into overlay B"). Composition then corresponds to performing one interaction after another, or linking outputs to inputs in a chain. Category theory insists that if you have a chain of transformations (say, $A \to B$ and $B \to C$), there is an unambiguous *composed* transformation $A \to C$ that is independent of how you parenthesize the operations. This matches a fundamental expectation for UIs: performing steps in sequence has a predictable outcome no matter how they are grouped. It also implies that a complex interaction can be abstracted as a single morphism – a crucial feature for managing complexity (think of bundling a multi-step workflow into one "macro" action).

Composable UIs as Categories of Components

Modern UIs are built from *components* that are meant to be reusable and composable. We can formalize a UI component (like a widget or panel) as an object in a category **UIComponents**. A morphism between component A and component B could represent **embedding** or **connection** – for instance, $A \to B$ might mean "A is embedded inside B" (as a child in a scene graph) or "A's output data flows into B's input". The exact interpretation can vary, but the key is that these relationships compose. If component A is embedded in B, and B in C, then A is effectively embedded in C. In category theory terms, if we have morphisms A → B and B → C, then their composition A → C exists and denotes the indirect embedding [6].

This simple idea has concrete echoes in UI frameworks. Consider **scene graphs** used in graphical applications: a scene graph is essentially a tree (or DAG) of objects where each node's transform (position, orientation, etc.) composes with its parent's transform. If we label each node as an object and each parent→child attachment as a morphism, the entire scene graph can be seen as a category (or a representation of one). The associative law corresponds to the fact that if Object A is inside B, and B is inside C, then the transform from A up to C is well-defined (and you don't need to worry whether you compute A→B then B→C, or directly A→C). In DonutOS's toroidal 3D interface, we indeed have a notion of nested or attached components: for example, **membranes** (floating panels) can dock to each other or to the main torus [7]. We could treat docking as a morphism that obeys transitivity. Under the hood, these docks and attachments form a graph structure that could be studied categorically.

Furthermore, category theory provides tools to reason about **modularity**. A well-designed category often has special constructions like **products** or **coproducts** (which generalize the idea of "and" and "or" combinations of components) and **initial/terminal objects** (which can model things like an empty layout or a universal container). For instance, a **monoidal category** is a category equipped with a binary operation (⊗) to combine objects with an identity element for that operation [8] [9]. If we consider a UI where multiple panels can be shown side by side, we could introduce a monoidal operation ⊗ meaning "display alongside" (or "overlay on top of" depending on context). **Monoidal composition** of UI elements would allow us to formalize the idea of an **overlay**: placing a glimmer effect on top of the torus could be seen as "Torus ⊗ GlimmerOverlay" producing a new composite object. In DonutOS's current spec, the system supports **glimmer and nimbus creative overlays** on the base torus geometry [10]. We can interpret the application of a glimmer or nimbus as a monoidal operation in a category of visual effects – where the torus (an object representing base scene) tensored with a "glimmer" object yields a combined scene. Monoidal categories require that combining UI parts is associative ($A \otimes (B \otimes C) = (A \otimes B) \otimes C$) and has an identity (some neutral overlay that does nothing). Ensuring these algebraic laws would mean, for example, that it doesn't matter in which order we apply multiple overlays; the final visual outcome should be the same – a desirable property for a predictable UI.

Reactive Programming and Functorial Time

Reactive programming is at the heart of modern interactive UIs. In a reactive paradigm, we think of UIs as continuously updating in response to streams of events or data changes (e.g. the position of a slider is a continuous stream of values; UI displays are functions of time). Category theory, especially through **functors** and **presheaves**, provides a potent way to formalize reactive systems. A **functor** is a mapping between categories that carries objects to objects and morphisms to morphisms, preserving identities and composition [11]. One classic example is viewing **time** as a category (imagine each moment or state as an object, and progression of time or event occurrences as morphisms between states) and then viewing a

reactive signal as a functor from the time category to the category of UI values. In other words, a time-varying value can be seen as a functor **F** from category **Time** to, say, **UIStates**, such that for each time-object t we have a UI state F(t), and for each "advance of time" morphism t → t' we have an updating function F(t) → F(t') in the UI [12] [13]. The functorial requirement means the UI update respects the composition of time intervals (if one interval is composed with another, the resulting state change equals applying the two corresponding UI updates in sequence) [11]. This is essentially the principle of *consistency over time*: no matter how you break up a time interval into sub-steps, applying all the small updates or one big update yields the same final UI – a commutative diagram in time.

A concrete instance of this idea appears in **Functional Reactive Programming (FRP)**. FRP defines *behaviors* (time-continuous values) and *events* (discrete occurrences) as first-class entities. Conal Elliott and others have described reactive behaviors as higher-order signals that can indeed be understood category-theoretically. In fact, one can treat each behavior type as a **category of states** and changes, where **objects are states** and **arrows are state transitions (changes)** [14]. The "no-op" change is the identity arrow (doing nothing keeps the state unchanged), and sequential changes compose to yield a resultant change [14]. This exactly matches the categorical axioms and gives a formal underpinning to *reactivity*: any reactive behavior provides not just values but a structured category of its evolutions. A **functorial** view comes in when we consider transformations of behaviors. For instance, a UI element that displays a value (say a label showing the current temperature) can be seen as a functor that maps an underlying **Data** category (where objects might be temperature readings, and morphisms are updates when the temperature changes) to the **UI Display** category (where objects are label renderings and morphisms are visual updates). **Functoriality** ensures that if a sequence of data changes is composed into one overall change, the visual updates compose to the same overall effect [11]. This property is critical for avoiding glitches: it guarantees that whether the data arrives in small increments or in a bulk update, the final UI is consistent.

Consider a more involved reactive scenario: DonutOS integrates sensor analytics (like brain-computer interface **BCI** signals, gaze tracking, etc.) with visual overlays [15]. These inputs can fluctuate rapidly and the UI must respond smoothly. We could model each sensor input as a category of its possible readings and transitions. Then the process of *fusing* BCI, gaze, and dwell (focus) signals into a single intention – which DonutOS lists as an input plumbing feature [15] – can be seen as a **functor (or a functorial construction)** that takes multiple input categories into a unified **Intention category**. For example, one might have a category **Gaze** (where objects = where the user is looking, morphisms = gaze shifting), a category **BrainSignal** (objects = decoded mental commands, morphisms = changes in mental state), and so on. The "fusion" would be some operation combining these; category theory suggests using a **product category** or **pullback** to combine synchronous events, and then a functor out of that product to the category of UI actions. Ensuring this mapping is functorial would mean that if the user's gaze and brain signals change in a certain pattern, the resulting UI action will be consistent no matter how we conceptualize the timing of those changes, as long as the overall pattern is the same. This is quite an abstract way to design an input handler, but it has practical payoff: it implicitly guarantees **consistency and order-independence** of integrated signals. In a safety-critical UI (like one controlled by BCI), this algebraic consistency can prevent race conditions or contradictory commands from simultaneous inputs. We will revisit safety in a later section, but suffice it to say that reactive systems greatly benefit from these categorical models – indeed, researchers have built entire categorical foundations for FRP [16] [14], finding that treating "changes" as morphisms clarifies how to compose and control them.

Model-View-Controller (MVC) through Functors and Natural Transformations

The **Model-View-Controller** architecture is a time-tested pattern in UI design. In MVC, the **Model** represents the core state and logic, the **View** renders the state to the user, and the **Controller** handles user input to update the model. While traditionally described in OO terms, MVC can be reframed in category theory. We can think of the **Model** states as objects in a category **M**, and user-driven state transitions as morphisms in that category. Likewise, the **View** can be seen as a category **V** of possible visual presentations (for instance, different UI widget states, or even a simpler category like "what text is shown on a label"). The connection from Model to View is essentially a **functor** F: M → V that maps each model state to a corresponding view state [17] [18]. The requirement that the view consistently reflects the model means that if the model transitions from state X to state Y (a morphism f: X→Y in category M), the view should transition from F(X) to F(Y) (that is, there is a morphism in V corresponding to that change). Functoriality precisely captures this: F(f): F(X) → F(Y) must equal the view's update corresponding to f. In other words, the diagram ModelX → ModelY (via f) and then functor F to View must equal doing F first and then the view's own change [11] – a basic commutativity condition guaranteeing that *view = F(model)* at all times. This is essentially a **naturality** condition if we also consider that the Controller provides a kind of inverse mapping (View events to Model updates); formally, one can model user input as another functor (from a category of UI events to model morphisms) and ask that the composition forms a **natural transformation** between appropriate functors. Natural transformations are the category theory way of saying "the two ways of mapping between these structures are aligned" [19] – which in MVC terms is alignment between user actions on the view and corresponding changes in the model.

It is insightful to note how some modern variations of MVC (and related patterns like MVVM or Elm Architecture) mirror these categorical ideas. For example, in purely functional MVC frameworks (such as Elm or certain Haskell libraries), the model is updated by pure functions and the view is a pure function of the model. This essentially treats the view function as a functor on the single object (the model state) or as part of a **bifunctor** if we include both model and UI as varying. Indeed, a Haskell library `mvc` described by Gabriel Gonzalez enforced that the **Model** was pure and separated, and used **functor combinators** to merge multiple controllers or views by mapping their outputs into a common type [20]. For instance, if you have two controllers producing different input types (say keyboard events vs. mouse events), you can **functorially map** (using `fmap`) those into a unified input type (like an `Either InputA InputB`) so that they can **compose together** [21]. This is exactly using a functor to reconcile two categories (each controller's event stream) into one common category of inputs [21]. The design of merging views or controllers via `fmap` and monoidal append corresponds to taking advantage of **functoriality and monoidal structures** to maintain MVC's separation while still combining components [20] [22]. Category theory concepts like **bifunctors** (functors of two arguments) appear here as well – for example, an MVC framework might treat the view as a contravariant functor in the model state (since it "consumes" the model to produce output) and the controller as a functor from input events to model updates; the entire application could be seen as an **adjoint pair** or some higher-order categorical construction linking these functors. Without diving into unnecessary detail, the key takeaway is: MVC's clean separation can be rigorously described by a network of functors preserving structure. This not only clarifies how MVC should behave, but can also identify generalizations – for example, one could mathematically reason about when two different UI architectures are equivalent by finding a natural transformation or equivalence of categories between their model/view behaviors [23] [24].

For DonutOS, which contains a **Primary/Unit Control membrane**, numerous analytical overlays (for manifold analysis, chaos, RL, etc.), and an **Intention Field** for user input [25], the MVC paradigm is

distributed: each membrane or overlay might have its own mini-MVC loop. Category theory could help coordinate these loops. We might treat each membrane's internal state as an object in a large product category of states, and each overlay's rendering as part of a big functor that maps global state to a composite view (the torus HUD). If done carefully, this functor would ensure that all membranes and overlays update in lockstep with the underlying **"state"** object that aggregates everything. In essence, DonutOS's entire UI could be seen as one grand Model-View functor, with various natural transformations representing user interactions funneling through. That guarantees coherence: for example, if an overlay representing "torus confidence" (a metric from manifold learning) monitors the model's data, a naturality condition can ensure that adjusting data via another panel yields the same effect as the overlay updating itself – so the **commutative triangle** between Model, Panel control, and Overlay display commutes. This is a form of **visual proof** that the system is consistent (we'll talk more about visual proofs via diagrams later). In summary, category theory doesn't replace MVC – it *explains* why MVC works and helps extend it safely to complex, multi-controller systems like DonutOS.

Overlays, Contexts, and Presheaves

One category-theoretic notion particularly relevant to UIs is the idea of a **presheaf**. A presheaf is a functor from a category C to the category of sets (or other structures), typically contravariant (meaning it reverses the direction of arrows) [12] [13]. In UI terms, presheaves can model data or interface elements that depend on context. Think of C as a category of *contexts* or *states of knowledge*, and the presheaf F: C^op → Set assigns to each context C an *information set* F(C) (e.g. the set of UI widgets visible in that context) and to each inclusion of context (say context refinement c1→c2) a restriction function F(c2) → F(c1) (e.g. how a UI in a larger context restricts to a smaller context) [12]. If that sounds abstract, consider a concrete example: DonutOS might have different **scenes** or modes – for instance, a "creative mode" vs "analysis mode", or simply different zoom levels of the torus (personal scale vs. group scale vs. global scale). We could define a category **SceneCtx** of scenes or contexts, where there is a morphism from context A to context B if B is a more *specialized* context than A (for instance, a hierarchy: Global context → Group context → Personal context, or Creative mode → base mode). A **presheaf on SceneCtx** could then assign to each scene context a collection of UI elements (the overlays/panels active in that scene) and to each transition (say from Group context to Personal context) a rule for which elements persist and how (the restriction map might, for example, hide certain overlays that only make sense at group level, or carry over certain elements with possibly modified parameters).

This presheaf perspective ensures **consistency across contexts**. The formal presheaf condition (that F(id_C) = identity and F(g∘f) = F(f)∘F(g) for composable context morphisms f, g) guarantees that if you move from context A to B to C, you get the same UI as going directly from A to C [26]. In practice, that could mean that if DonutOS transitions through multiple intermediate scene states, the resulting set of overlays is well-defined and independent of transition path. It's a way to prevent, say, a widget from erroneously lingering when it shouldn't or disappearing if you toggle modes in a different order. All possible "restriction" or "extension" maps between contexts commute.

Another application of presheaves is in **data availability and overlays**. Some overlays (like analytics) might only have meaning when certain data is present. We can treat the set of available data dimensions as a category (with inclusion as morphisms), and a particular overlay as a presheaf that to each data subset assigns a set of possible visual representations. For instance, an overlay that plots a manifold confidence might require the presence of "manifold features" in the data. If those features are absent, the presheaf yields an empty set (the overlay cannot exist). If features are present and then more features are added,

there is a restriction map from the richer context to the smaller one which could, for example, project high-dimensional features to those the overlay knows about. In categorical UI design, these ideas relate to **sheaves** and **presheaves** ensuring local consistency of information – a powerful concept for AR interfaces where not all information is globally available at once. Although we won't dive deep into sheaf theory here, it is tantalizing to note that **don't panic**: category theory has even tackled the challenge of "gluing local views into a coherent global view" via sheaves. This resonates with DonutOS's principle of "explicit ambiguity where data are thin" [27]. One could imagine a sheaf that captures user's beliefs or sensor data across regions of the torus, explicitly showing ambiguity where overlapping regions don't agree, leveraging **presheaf** assignments and their failure to be sheaves as a visual indicator of uncertainty. In short, presheaves give us a structured way to talk about UI elements spread across overlapping contexts and ensure they're glued together correctly when contexts intersect.

In summary, category theory's core concepts – **objects/arrows, composition, functors, natural transformations, monoidal operations, and presheaves** – each illuminate an aspect of UI design:

- Composition (the essence of category) underpins the **composability** of UI components.
- Functors capture **mappings** between different layers of abstraction (e.g. data to view, model to view), preserving consistency of structure.
- Monoidal categories and related algebraic structures formalize the combination of independent UI elements (like overlays or side-by-side panels) with identities (empty overlay, empty panel).
- Presheaves model context-dependent UI pieces and ensure consistency when navigating contexts or modes.

Using these tools, we can now turn specifically to **DonutOS** and analyze its current architecture through this lens. We will see that many features the Donut Spec highlights – membranes, overlays, scenes, analytics – can be naturally described in categorical terms, which not only validates the design but also suggests generalizations. This will set the stage for forecasting how DonutOS could evolve: introducing new algebraic "glue" and metaphors for a truly next-generation UI that is as rigorous as it is imaginative.

Categorical Analysis of DonutOS's Architecture

DonutOS is described (in the project's spec snapshot) as a modular system composed of **membranes** (docked or floating panels), geometric primitives (a central torus with layered overlays), and various subsystems for analytics, input, and scenes [25] [28]. Its design principles emphasize *coherence over control*, *visible states*, and *safety via explicit ambiguity* [29] – all of which align well with a categorical mindset of making relationships explicit and ensuring operations have well-defined outcomes. Let us walk through key aspects of DonutOS and recast them categorically:

- **Membrane System (Dock/Floating Panels):** DonutOS's membrane system allows users to "compose, dock, and float membranes (panels) that control geometry, intention, analytics, and creative overlays" [1]. Categorically, we can think of each **membrane** as an object in a category **MembraneUI**, and a **dock relation** as a morphism. If membrane A docks onto membrane B, we have a morphism $A \to B$ (perhaps also one in the opposite direction if the docking is two-way). These docking relations likely form a hierarchy or graph (for instance, you might dock multiple panels into a tabbed interface, etc.). The interesting categorical structure here could be a **partial order** of panels (if docking is hierarchical) or a more general directed acyclic graph. Composition of dockings means if A is docked into B and B into the main Torus HUD, then A is effectively docked into the Torus.

Ensuring associativity of composition means the end result is independent of docking order – which in practice is true: no matter the sequence in which you attach panels into a stack, the final nested structure is what matters.

Moreover, docking panels could be modeled as a **pushout** operation in category theory terms (which we will expand on later). Briefly, if two panels share a common interface element (say a handle or data context Z), "gluing" them together along that interface is a pushout in the category of UI layouts [30]. For now, within the membrane system, consider that each panel has an **input interface** (what data or control it expects) and an **output interface** (what data or effect it produces). Docking two panels typically involves connecting the output of one to the input of another (for example, a panel that selects a data source docked to a panel that visualizes that data). This is very much like composing two morphisms: Panel A → Data (some data stream), and Data → Panel B (where Panel B expects that data). The composite is Panel A → Panel B through that data channel. Ensuring **type compatibility** of docking can be viewed as ensuring the existence of a morphism in the category: if Panel A's output type and Panel B's input type don't match, there is no morphism and the composition (docking) is not allowed. Category theory thus explains when membranes can compose: they must share a compatible intermediate object (data or control line) to glue along. In DonutOS, a Membrane Directory/Builder is noted [31] – one could imagine it uses a schema to decide which panels can dock. Schema matching is a category theoretic idea of finding a **span** or **pullback**: two objects share a sub-object Z, so you can pull back along Z to join them. We see here the seeds of a formal approach to membrane composition that we will fully articulate in the "gluing tools (pushouts)" discussion.

- **Torus Geometry & Overlays:** The centerpiece of DonutOS is a 3D torus representation of attention, with various rotation modes and overlays (glimmer, nimbus, creative overlays, "levogyre" shells, etc.) [10]. The torus itself can be thought of as an object in a category **Scene**, representing the base scene graph. Overlays are additional objects that modify the scene. We previously mentioned thinking of overlay application as a **monoidal operation** (Scene ⊗ Overlay → Scene'). Indeed, DonutOS lists an upcoming feature: **Grids/Platonic overlays**, which would allow layering multiple grid structures or Platonic solid patterns on the torus [32]. Each overlay (be it a grid, a Platonic solid outline, a glimmer effect, or a "Solar Gate" sun symbol) can be seen as an endomorphism on the scene object – i.e., a morphism TorusScene → TorusScene that adds that particular visual augmentation. If these overlays function independently, they likely commute or can be applied in any order, suggesting that the collection of overlays forms something like a **commuting family of endomorphisms**. In categorical terms, if each overlay is an arrow `overlayX: Torus → Torus` (conceptually a state transformer on the scene), then ideally `overlayA ∘ overlayB = overlayB ∘ overlayA` for different overlays (commutativity) and each has some idempotence or invertibility (applying the same overlay twice might be a no-op beyond the first application). This begins to sound like an **abelian monoid** of overlays under composition – which is a very structured situation. Alternatively, we model the application of multiple overlays as a **coproduct** (taking the disjoint union of effects) or as elements in a **power set** lattice of possible overlays. Either way, category theory provides tools to formalize how multiple effects combine.

Another aspect: the torus has **rotation modes** and **gyro/levogyre** inputs [33]. Rotation symmetries of a torus form a mathematical group (a continuous group, essentially SO(2)×SO(2) for rotations around two independent axes). We can incorporate this by noting that the scene category likely has an automorphism group acting on it (the group of torus rotations). Category theory doesn't force us

into group theory, but it's common to study categories with group actions or to build **quotient categories** by an equivalence relation. If rotation is a reversible operation (no net effect if you rotate fully around 360°), that can be treated as an isomorphism in the category of scenes. A **morphism** that is an isomorphism (invertible) can be considered a symmetry operation. Thus, the torus rotations and flips (there is mention of "symmetry flips" [10]) generate a subgroup of the automorphism category of the scene. By modding these out, one could consider an attention state up to rotation (i.e., focusing somewhere on the torus ignoring where zero-angle is) as an equivalence class in the category. This viewpoint might help in ensuring the UI is invariant under certain operations (if intended). For example, maybe the *Intention Field* overlay is meant to be rotation-invariant (so its meaning doesn't change if the user rotates the torus); one could enforce that by requiring certain morphisms commute with the rotation isomorphisms, which is the idea of an **equivariant functor** (preserving symmetry) or a **natural transformation** that ties a rotation to a corresponding state change.

The mention of **levogyre shells** and **Solar Gate (Sun symbols)** [10] in the spec evokes a layered concentric structure around the torus (shells) and possibly markers for orientation (sun symbol gates). We can imagine modeling these as additional *layers* in the scene graph. Picture an onion-like series of shells around the torus, each shell rotating at some ratio (the spec references a "phase lock + ratio detector and 'bindu window'" in extensions [28], which suggests aligning rotating shells according to rational ratios – reminiscent of frequency locking in oscillators). Each shell could be an object (like Shell1, Shell2, …) with a morphism to the torus indicating it's attached/co-axial. The alignment of shells (phase locking) could be seen as requiring that certain morphisms (rotations of shell vs rotation of base torus) satisfy a **commutative diagram** – essentially a diagram expressing that rotating the torus by some amount equals rotating a shell by a corresponding amount (if locked). In diagram form: TorusState --rotate--> TorusState (some angle) and going to ShellState then rotating should equal going to TorusState then to ShellState. That commuting square would visually prove the shell is locked to the torus (no slipping). If out of lock, that diagram doesn't commute and the difference is the phase drift – which actually could be visualized as a path around the torus vs shell that doesn't close. It's fascinating that category theory can describe even these subtle relations: essentially shells and torus form an interconnected system that can be described as a **pullback** or **fiber bundle** (shell being a fiber that can either rotate freely or be fixed by a relation). For our purposes, it's enough to note that **commutative diagrams** (the bread-and-butter of category theory) can encode synchronization conditions like these. We will use this idea later when discussing *visual proofs*, but it's already implicit in how one might ensure consistent overlays – e.g., the Solar Gate symbols might appear aligned on every shell when a certain phase condition holds, which in diagram terms is a big commutative diagram of all shells and a reference frame.

- **Analytics and State Functors:** DonutOS includes a suite of analytics overlays – mean-field thermodynamics, connectome graphs, manifold confidence, predictive coding, etc [34] – each presumably taking data from the user's context and presenting a visualization or interactive control. These can be thought of as **functors from a data category to a visualization category**. For example, the **manifold (torus confidence) overlay** likely computes how well the user's data or attention patterns fit a torus manifold [34]. We can model the underlying data (perhaps a point cloud of user's cognitive state features) as an object D in a category **Data**. The manifold scanner that yields a "torus confidence" value is then a morphism in a processed data category (perhaps **Analysis** category). Finally, the overlay that displays a colored indicator or number for torus confidence is a morphism from that analysis result to a UI representation (in **View** category). Composing these, we

have Data → Analysis → View. We could combine these steps into a single functor (or two sequential functors) from Data to View. A categorical benefit is that if multiple analytics share underlying data transformations, we can factor those out as functors to common sub-results. For instance, both the connectome graph overlay and the complexity overlay might need the graph of user connections. Instead of each computing it separately, we have a functor F: Data → NetworkGraph and then one functor G: NetworkGraph → View (connectome overlay) and another H: NetworkGraph → View (complexity metrics overlay). By functorial composition, the overall pipeline is clear and avoids duplication. **Natural transformations** could represent different ways of computing the same metric (maybe a heuristic vs an exact method) and category theory assures us that if they are truly the same, there should be a natural transformation connecting those functors (ensuring results are isomorphic). This level of formalism might be overkill for UI metrics, but it hints at something important: *state consistency functors*. All overlays and panels in DonutOS ultimately depend on the user's central state (the state can be thought of as a big tuple of geometry, intentions, data streams, etc., stored in `state` as noted with a localStorage key [35]). Each overlay essentially presents a *view* of that state or a subset of it. We should have functors F_i for each overlay i: **State → $\mathbf{OverlayState_i}$** (and further to the actual rendered view). State consistency then means: if the underlying state changes via some transition (a morphism in State category), each overlay's functor F_i will map that to a corresponding morphism in its view, updating that overlay. If multiple overlays rely on the same part of state, then functoriality and having a shared source means they stay in sync by construction (they're all reading from the single source of truth). In an imperative system, this synchronization is hard; in a categorical design, it's a straightforward consequence of using a **universal state object** and functors for projection. This approach can also formally manage **constraints**: for example, if overlay A and overlay B both adjust a shared parameter (say brightness), there should be a natural transformation between their functors or a mediating morphism ensuring that adjusting via A or via B yields the same global state change (commutativity again). The spec's UX principles "make states visible" and "clear resets" [36] align with this – a categorical model would explicitly surface state as objects and arrows, making it easier to implement visible state indicators and global reset morphisms that bring you to an initial object.

- **Scenes and Immersive Modes:** DonutOS supports **Scene presets** with cycling and is planning immersive staging [37]. A scene can be considered an object in a category **Scenes**, as discussed under presheaves. Likely, switching scenes means swapping out a whole set of overlays, color schemes, audio, etc. Category-theoretically, a scene change could be a morphism that changes the current UI context (e.g. a pushout that replaces one set of panels with another, or a simple state morphism that toggles certain flags). The **Scene Algebra** idea (which we will detail in the next section) treats scenes as algebraic objects that can combine or transform. Within the current architecture, scenes might be independent (one active at a time). But we can still formalize the act of scene switching as an arrow S1 → S2 in a scene transition category. Composition of such arrows yields multi-step transitions or cyclical rotations. If scene changes are cyclic (the spec mentions keyboard cycling through scenes [37]), then we effectively have a category where each scene is connected in a cycle – that's isomorphic to a cyclic group action on the set of scenes. Recognizing that, we might design the scene cycling feature by imposing that the "Next" operation is invertible (so there's a Prev that undoes it, making scene selection a group or groupoid action). This prevents, say, going Next five times and Prev five times from landing you in a different state – a consistency condition any user would expect.

In summary, examining DonutOS piece by piece reveals that **it is already structured like a categorical system**, even if informally:

- There is a **universal state** (or a structured collection of state components) that many parts of the UI depend on – which invites a functorial "Model → View" interpretation for each part.
- Membranes and overlays are **modules that compose**, suggesting pushouts (for docking) and monoidal compositions (for layering) as the categorical operations managing their combination.
- Scenes and modes form a **context space** that can be formalized as a category or group, with UI elements as presheaves over that space, ensuring consistency when moving between modes.
- Complex interactions like phase-locking shells or synchronizing multi-input control can be captured by **commutative diagrams**, providing *visual proofs* of consistency (e.g., demonstrating that two different user manipulation paths lead to the same final configuration by showing the diagram of transformations commutes).

By grounding DonutOS's present architecture in these terms, we not only validate its design with proven abstract principles, we also uncover the "slots" where new categorical constructs could enhance the system. This paves the way to our next section: **Speculative Future Directions**, where we will use category theory not just to describe but to *drive* new features for DonutOS. We will introduce concepts like a formal **Scene Algebra**, a **node-graph programming interface** for overlays, the use of **pushout gluing** to modularly extend the UI, **state consistency functors** to maintain coherence across subsystems, and **visual proof diagrams** as part of the UX for transparency and trust. In doing so, we keep one foot in the rigorous world of category theory and one in the imaginative ethos of DonutOS – invoking metaphors of Platonic solids and fractal toroids to inspire, while relying on algebra and topology to implement safely.

Future Directions: A Category-Theoretic Roadmap for DonutOS

With a solid understanding of how category theory underpins current features, we can boldly speculate on **future directions** for DonutOS – essentially, how we might consciously *engineer* the next generation of DonutOS using category theory as both toolbox and compass. These proposals are a mix of rigorous design and imaginative leap, fitting for DonutOS's blend of *scientific* and *mythic* spirit. Each subsection below outlines a visionary idea and ties it to concrete categorical constructions:

1. Scene Algebra: Composing and Transforming Attention-Scenes

In DonutOS today, scenes are presets or modes – but we can imagine a **Scene Algebra** where scenes are first-class objects that can be manipulated, combined, abstracted, and even mathematically reasoned about. What would this entail? First, define a category **Scene** where each object is a "scene configuration" (a particular arrangement of the torus, lighting, active overlays, audio ambiance, etc.), and a morphism between scenes is some transition or transformation (for example, a smooth interpolation, or a discrete switch that turns on/off certain overlays, or a move from one environment to another). If we endow this category with additional structure, it can become an algebra.

One idea is to define a **monoidal operation** on scenes, denoted ⊕, that represents a sort of *merging* or *superposition* of two scenes. If scenes were simple sets of overlays, ⊕ could be their union (with some resolution if there's conflict). More fancifully, ⊕ could represent placing two scenes side by side in split-screen or layering one scene's visual theme on another's data content. For example, imagine "Scene A: creative (with glimmer overlays and musical feedback)" and "Scene B: analytical (grid overlays, numerical

displays).” A possible ⊕ could yield a scene that takes the *content* of B but the *style/overlays* of A – a blend mode. In categorical terms, if we have **projections** (functors) that extract the style aspect and the content aspect of a scene, ⊕ might be constructed via a **pullback** or some universal property that aligns the content of one with the style of another, yielding a new scene that commutes appropriately (so that projecting this new scene to style gives A’s style, and projecting to content gives B’s content). This is speculative, but category theory provides a disciplined way to formalize it: we could say the new scene is a **limit** in a diagram where it has arrows to A and B that preserve respective properties. In plain language, the algebra would allow a user or developer to say: “Take the lighting and fractal background from scene X and apply it to scene Y’s geometry and panel layout” – if such an operation is defined, we know from category theory that it should satisfy certain laws (like associativity if you do multiple merges, existence of an identity scene that does nothing, etc.). One might designate a “blank scene” as identity for ⊕ (which if ⊕ is union, identity is the empty scene). Ensuring this identity and associativity can make complex combinations tractable: the user could combine scenes in any grouping and get a consistent result.

Another algebraic aspect is **transitions**. In conventional terms, one might have a library of scene transitions (crossfade, spin the torus during transition, etc.). Category theory suggests looking at transitions as **morphisms** and possibly enriching the category with a notion of **2-morphisms** (morphisms between morphisms) or a path composition algebra. But likely, keeping it simpler: If we know how to compose scenes via ⊕ and perhaps how to *invert* scenes (some scenes might have an “opposite” or a way to undo their effect), we can generate transitions by difference. For instance, given scenes A and B, perhaps one can formally define a transition morphism τ: A → B as “apply (A^{-1}⊕B)” in some gradual way. If A^{-1} represents the inverse of A’s settings (the changes needed to neutralize scene A back to baseline) and then ⊕B adds scene B’s settings, τ can be conceptually broken into two phases: remove A, add B. If these operations commute or can interleave, a smooth transition is possible. The algebra could specify which elements of a scene can change independently, enabling simultaneous crossfade of audio and cross-dissolve of visuals, etc. This is all to say: a **structured algebra of scenes** would allow DonutOS to not just switch modes but *blend* them, script them, and even prove properties about them (like commutativity of applying two independent scene changes).

Such an algebra can also aid **safety** and **consistency**. If scenes are algebraic objects, one can define **invariants** (certain overlays that should always remain across specific combinations, or limits on brightness sum, etc.) as algebraic constraints. Category theory, especially via equational logic or topos theory, can enforce invariants by restricting to subcategories where those invariants hold (e.g. a subcategory of Scene where “Keep overlays/toggles bounded for comfort” [38] – an invariant to not exceed brightness/clutter thresholds – is built-in). Then any composition of scenes in that subcategory automatically respects those comfort bounds, by closure of composition in the subcategory.

Finally, a Scene Algebra could incorporate the **mythic metaphors** of DonutOS: consider making Platonic solids (cube, tetrahedron, etc.) part of the scene composition vocabulary. Perhaps each Platonic solid overlay (as planned [32]) corresponds to a transformation or symmetry in the scene algebra (e.g. imposing a cube grid might align the scene with 90° rotations – linking to a dihedral group action within the category). The *levogyre shell* might be an element in the algebra that one can multiply a scene by to get a rotated version (like a generator of a cyclic subgroup). By giving these imaginative elements algebraic roles, we allow the user to play with them in a consistent way – almost like a sandbox of sacred geometry: you could “multiply” a scene by a dodecahedron overlay and know exactly how it will transform the interactive degrees of freedom (perhaps quantizing rotations to 30° steps, etc., since a dodecahedron has 12 faces, etc.). While this is speculative, it shows how category theory can even integrate the symbolic layer of design (Platonic

solids as symbols of order) with concrete interactive functionality (overlays imposing certain structures), all under a unified algebraic structure.

In summary, a **Scene Algebra** would give DonutOS a powerful, flexible way to generate new modes and mixed realities on the fly, while guaranteeing consistency through algebraic laws. It invites users to *compose their own reality* in a safe sandbox – almost a creative toolkit where the *laws of UI physics* (associativity, identity, commutativity of independent effects) hold, so users can predict outcomes even of complex combinations. It turns scene management into a science of patterns, aligning with DonutOS's holographic principle by letting small modular changes reflect across the entire experience fractally.

2. Node-Graph Overlays: Visual Functional Programming for UI

DonutOS emphasizes an exploratory yet safe UI, and one way to empower advanced users (or developers) is by exposing the system's compositional structure through a **node-graph editor**. Many modern creative tools and game engines offer node-based visual scripting, which aligns perfectly with category theory: a node graph is essentially a pictorial representation of a **category or a diagram** in a category, where nodes are objects and connections are morphisms. We propose **node-graph overlays** where the user can inspect and modify the connections between DonutOS components in real-time.

Imagine an overlay that, when activated, fades the torus scene slightly and overlays a graph of nodes and arrows representing active data flows and control links in the UI. Each panel (membrane) or overlay becomes a node; each significant relationship (like "Panel A's output feeds Panel B's input" or "Overlay X is triggered by Scene Y's activation") is drawn as an arrow. The user can literally see the category of their UI instance. This would be a kind of *interactive commutative diagram*: if two panels provide input to one overlay, you might see a diagram shaped like a "Y", indicating a functor that merges two inputs, etc. The user could then rewire things: drag a connection from one node to another to re-route outputs (if type-compatible). Essentially, DonutOS could have a **user-accessible functor mapping** – by rearranging nodes, the user is re-defining which functors (or natural transformations) connect components.

For example, suppose DonutOS has a **connectome graph overlay** that currently visualizes the user's social network data. In the node editor, that overlay node would have an incoming arrow from a "Social Data" node. If the user wanted to repurpose that overlay to visualize a different network (say, conceptual links between their notes), and if the system allows it, they could detach the arrow from Social Data and connect a "Notes Graph" node into the connectome overlay node. Immediately, the overlay now draws the notes network instead, because we have functorially remapped its input. Under the hood, this is like setting a different functor F': NotesData → GraphOverlayView in place of F: SocialData → GraphOverlayView. As long as both data types can be seen as a graph, the overlay code might be generic. This is akin to function composition in programming, but done visually. It leverages category theory because the safe reconnection of nodes requires knowing the domain and codomain of morphisms (the "type" of data flows). The UI can enforce only valid compositions (just as only compatible types compose in programming). Perhaps each connection in the node graph is labeled by the data type or interface it carries. In category terms, these could be **hom-sets** constraints: there is a hom-set from SocialData to GraphOverlay only if SocialData has a graph structure the overlay expects. The node editor can consult a type inference system (backed by category-theoretic type theory) to decide which connections are allowable.

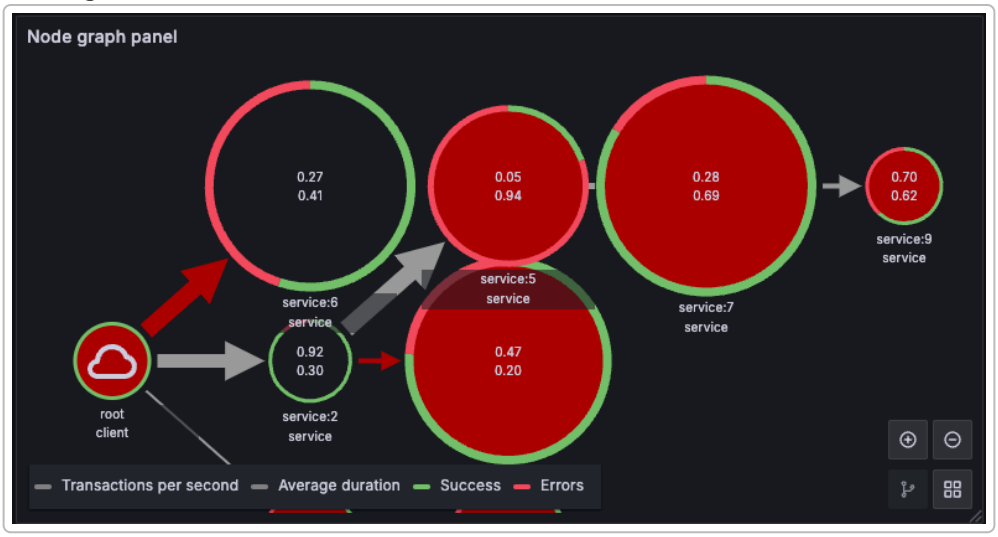
By giving users a node graph interface, DonutOS becomes **end-user programmable** in a highly visual way. This aligns with the idea of **Reactive functional programming** – but instead of writing code, the user

draws it. Each node can also be visually annotated with its state (since DonutOS values making state visible [39]). For instance, a node representing a filter might display the current filter criteria as text, which the user can click to edit. In effect, the node graph overlay is like exposing the otherwise hidden "wiring" of the UI, turning DonutOS into a *live dataflow programming environment*. Experienced users could chain analytics: e.g., drag an arrow from the "RL (reinforcement learning) overlay" node's output (maybe it produces recommended actions) into a "Journal logging" node to record recommendations when certain events happen. If the internal architecture has that modularity (and a reading of the spec's extension on CTI and journaling suggests it might [40]), then a node editor could reveal those modules and let the user connect them.

Critically, because the system is categorical (or can be designed to be), we can ensure that these user-made modifications preserve safety and consistency. The node editor essentially manipulates the **diagram of functors and transformations** that make up the UI. Category theory can formally verify that the resulting diagram still commutes where it needs to (for example, any essential invariants are maintained). One could implement a simple check: if a user tries to do something that would break a commutative diagram (e.g., feed inconsistent sources to something that expects a single source), the system can highlight that in red or prevent it. In node terms, this might mean you can't have two different active controllers driving the same model input unless you explicitly insert a merge node that resolves them (which itself corresponds to adding a morphism that merges – which the system might provide as a combinator node, analogous to an **either/or** merge or an averaging node, etc.). In more advanced terms, one could incorporate **typed lambda calculus** or **profunctor** models to allow creation of new combined behaviors. But even with simple directed acyclic graphs, a lot is possible and user-friendly.

From a **UI mythos** perspective, a node-graph overlay is like revealing the *Indra's Net* of the interface – the web of connections that holds the torus of attention. It could be visually styled to fit DonutOS's aesthetic: perhaps neon lines and geometric node glyphs hovering over the torus, with slight levogyre swirling motion to indicate activity flows. The user becomes a weaver of their attention web, which is a powerful metaphor: they are not just a user but a co-creator of the experience, tweaking the fundamental relationships in their cognitive toolkit.

To ground with a reference example, consider Grafana or other flow-based tools that show node graphs for systems monitoring. For instance, Grafana's node graph visualization (see figure) shows nodes as circles with stats and edges between them



. In our context, nodes could similarly show real-time stats: a panel node might display its refresh rate or data metric, an overlay node might display a current value it's highlighting. The edges might be animated pulses if data is streaming (like how some node editors animate flow during execution). This provides intuitive feedback: if a pipeline is congested or slow (like a heavy analysis overlay), the pulse might blink slowly or change color, alerting the user in the node view that this part of the system is lagging. In this way, the node graph overlay also serves as a **debug and performance monitor** for power users. Category theory's influence here is subtle but present: by modeling dataflow as morphisms, the system can measure each morphism's performance (like a functor's mapping taking X ms) and annotate the arrow with it. This is akin to attributing a "weight" or metric to morphisms, which could be formalized as a functor into a category of metrics, but practically it's just instrumentation.

In summary, **node-graph overlays** would transform DonutOS into a living, user-editable diagram of itself. It empowers users to customize and extend the interface logic without writing code, simply by exploiting the **compositional, graphical nature of category theory**. It demystifies the system by providing a clear visual of "what is connected to what" – aligning with the principle of coherence and explicitness. And thanks to categorical constraints (types, invariants), it can guide users to make only valid connections, maintaining system integrity. This idea strongly complements the next idea of **gluing via pushouts**, as node editing is essentially the *manual* way a user could perform gluing of modules on the fly.

3. Gluing Modules with Pushouts: Structured Extension of the UI

As systems grow, one frequently needs to **extend** or **integrate** modules – for example, adding a new overlay or combining two tools into one interface. Category theory offers a specific construction for the general notion of "gluing": the **pushout**. A pushout formalizes how to join two objects that share a common part [41] [30]. In UI terms, think of two modules (panels, overlays, data pipelines) that have some shared interface or data structure. The pushout is essentially the operation of taking the union of the two modules, identifying (gluing together) their common interface so it becomes one.

Consider a concrete scenario: DonutOS currently has a **Membrane Directory/Builder** and separate specialized membrane panels [25]. Suppose a developer (or advanced user via node graph) creates a new panel type – for instance, a **"Phase Sync Control" panel** that allows fine control of the phase locking mechanism (given phase sync is an active extension [28]). They want to integrate this panel so that it works with the existing **Phase Oscillator Playbook** and the **Intention Field**. All three might share a concept of "phase oscillator parameters". In category terms, let's say Panel P (new Phase Control panel) exposes an interface Z = {frequency, amplitude} controls for the oscillator. The existing Intention Field panel Q also can adjust "phase amplitude" as part of user intention presets, which overlaps with Z partially (say it only touches amplitude). The pushout of P and Q along their common part (amplitude control) would glue them so they refer to the *same* amplitude setting rather than separate ones. In doing pushout, we identify P's amplitude control with Q's amplitude control, meaning any change in one is a change in the other – they're literally the same underlying variable now. The result P ⨿_Z Q is a combined panel that perhaps has both sets of controls but one amplitude slider controlling both. In practice, pushouts allow the system to **merge state and UI of modules without double-counting or conflict**, as long as we clearly define their shared sub-module.

Visually, one can think of pushout gluing as overlapping two diagrams on their common part. For instance, if one panel's output goes into a data filter, and another panel's output goes into a similar filter (same type), one might pushout at the filter: meaning use a single filter for both, feeding both outputs into it. This kind of refactoring or merging can be handled by a generalized **drag-and-drop** in the node editor: dragging one node onto another of the same type to indicate "merge these". The system would then compute the pushout behind the scenes – basically unify the node (like unifying two variables in logic). The conditions for a pushout require that the shared interface is consistent and that both modules map into it. If so, the pushout exists and is essentially their **amalgamated module** [42].

Why is pushout important? It provides a **universal solution** to the gluing problem, meaning if any combined system is to contain both modules and keep their shared interface unified, the pushout is the minimal such system (any other solution factors through it) [41] [42]. For DonutOS, this means the pushout gives a *canonical way to integrate modules* that ensures no redundant duplication of the common functionality. In practical terms, this could reduce bugs (by not having two separate amplitude states for two panels) and improve user experience (both panels always show the same value for the unified thing).

To illustrate with a simpler analogy: imagine two spreadsheets that both have a column "Date". A pushout along the "Date" column would merge them into one bigger spreadsheet aligned by Date, rather than having two separate date columns. Similarly, in UI, pushout might align timelines or coordinate systems between overlays. If overlay A and B both have their own timeline slider, gluing them on the timeline would produce a single timeline controlling both in sync. This is very relevant to something like combining an **immersive scene** overlay with a **serendipity journaling** overlay (as per extensions [40]). If both have a notion of time phase or cycle, gluing them on that ensures one timeline for user to scrub through combined history, rather than two.

One can also use pushouts to **incrementally build the UI**. In Donut Spec, new features are often mentioned as planned modules (like grid overlays, group sync, etc.) [43]. Using pushouts, the development could be modular: design the new feature in isolation with an interface to existing ones, then push it in. For example, group phase synchronization (syncing multiple users' donuts) might share the "phase lock metric" interface with the individual phase lock. The group sync module has its own logic, but to integrate it with a single-user UI, one glues on the phase-lock interface. The result is an extended category of UI elements

now handling both individual and group locks with the same visual indicator perhaps, thus maintaining coherence. Mathematically, if *UserDonut* and *GroupDonut* share some boundary conditions interface (the spec mentions boundary visibility and locks [44], which might become a shared concept if two donuts connect), then the combined system via pushout ensures that the *boundary conditions status* is one unified thing visible to the user, not fractured between modes.

The **visual proof** of a pushout's correctness is a *commutative square* [41] [19]. Displaying this to the user (especially a developer-user in the node view) can help explain what was glued. For example, after merging two nodes, the UI might briefly highlight the shared node in a distinct color, indicating "these were identified as one". If the user hovers, it might show a square diagram: P → Pushout ← Q, with P and Q's interface mapping to the unified one, to reassure that indeed both now feed into the same sub-part (this could be hidden under expert settings, but it's a nice transparency measure for those interested). It's essentially showing the **universal property**: any update from either side goes through the common part to affect the whole, which is exactly what we want (gluing ensures changes propagate).

From a mythic viewpoint, pushouts are like **alchemical marriages** of interface elements: two separate forms meeting at a common essence to create a new unified form. In the attention-torus metaphor, one could liken it to merging two rings through a common cross-section – creating a chain. A levogyre visual effect might represent the gluing point swirling together the colors of each module's aura. This creative framing aside, the functional benefit is clear: **structured extensibility**. DonutOS can grow feature by feature without becoming a tangle, because each integration is done via an algebraically sound operation (pushout) that guarantees minimal interference and maximal sharing of what should be shared.

4. State Consistency Functors and Invariants

As systems integrate more parts (BCI inputs, gaze tracking, multiple overlays, group sync, etc.), maintaining a single coherent state – or at least a consistent relationship between states – is paramount. Category theory provides the notion of **functors** to relate state spaces, and **invariants** as properties preserved by functors or morphisms (often captured by kernel/cokernel or equalizer/coequalizer constructions). We envision using **state consistency functors** to automatically sync and check different parts of DonutOS's state.

For a simple example, consider the relationship between the **3D torus rotation state** and the **2D projection on screen**. There is a natural **projection functor** F from the 3D scene state category to a 2D UI representation category (essentially the rendering pipeline acts like a functor: 3D transformations get mapped to 2D transformations of drawn elements). For consistency, if the user rotates the torus via a 3D control or via a 2D interface element (say an on-screen knob that corresponds to rotation angle), both should result in the same final orientation being displayed. This is a commutative diagram between the model change and view change [19]. Ensuring this means treating the mapping from model to view as functorial (so rotation composition in 3D corresponds to composition of the resulting view transforms, which a correct graphics pipeline does [11]). If any discrepancy arises (perhaps due to a bug), it would break functoriality – something we could detect if we instrument the system to compare the two paths (like if rotating 90° then 45° yields a slightly different pixel result than 135° once, that indicates inconsistency). A categorical approach could incorporate a **monitor** that checks certain diagrams for commutativity at runtime as an integrity assertion. For instance, we might tag critical sub-diagrams of the UI (like model->view->user input loop) and periodically verify the end-to-end consistency (like does the value shown equal the value internally, etc.). This could catch UI drift or mis-sync issues early.

A more complex consistency to maintain is between **user intention and system state**. DonutOS has an Intention Field (for BCI/gaze input) and presumably the actual application state of that intention (like an intended action vs executed action). By modeling user intention as one category (perhaps stochastic or fuzzy states from BCI) and executed state as another (concrete UI actions taken), we likely have a functor from the intention category to the action category (the system interprets intentions into actions). We want a form of *adjointness* or *pseudo-inverse* perhaps – i.e., that when the system does an action, it aligns with the user's intention in some optimal way (no surprise). We might formalize an invariant: **Intention Realization Invariant**: The composed mapping of "user's mental state → UI response → user's mental feedback" should ideally form a contraction (in a cognitive sense) – basically, the user's mental model and the UI's state should converge. This is abstract, but one way to enforce a piece of it is to have the UI reflect back the interpreted intention obviously (like highlighting what command it thought the user invoked). Categorially, that reflection is a functor going back from Action to a *predicted intention* (basically a left or right adjoint of the intention interpretation functor if one exists). If the composition of user intention to action to reflected intention is naturally **identity** (or close), we have a fixed point, meaning the system correctly understood the user (the user sees the UI did exactly what they meant). If not, there's a discrepancy. A UI could highlight that ("Did you mean…?" dialogs). This is reminiscent of **paraconsistency** notes in the spec [44] – dealing with ambiguity and multiple hypotheses. Category theory can deal with that via **sheaves of possibilities** or branching categories, but here we focus on invariants: ideally one invariant is "no unnoticed ambiguity": if the system is unsure what the user intends, it must explicitly query or show both possibilities (DonutOS principle: show ambiguity explicitly [45]). This principle can be encoded: if a functor F: Intention → Action is not one-to-one (not invertible), the system must spawn an *ambiguity object* capturing the preimage choices (this could be modeled as a **pullback** from Action to a multi-valued Intention outcome). Invariant: There should be **no commutative square** where one side is user → ambiguous actions and the other is user → chosen action unless an ambiguity object is present to break the square. This is technical, but essentially ensures you never silently pick an arbitrary action when intention mapping is many-to-one. The UI will instead create a branch (like a prompt). This idea comes straight from ensuring certain diagrams do *not* commute unless user confirms a path – which ironically is using category theory to enforce an anti-commutativity in those cases (or a controlled one).

Another state consistency area: **Group synchronization**. If multiple users link their DonutOS instances (the spec hints at group phase sync and perhaps future collaboration [40]), we get a distributed system. Category theory handles distributed state via concepts like **span of functors** or **synchronization via limit/colimit**. A group of N users could be considered as N state functors into a joint state (a product or pullback if fully synchronized, or some loose coupling if partial). Consistency means if one user changes a shared scene property, all others' views reflect it. That is ensured by having a **broadcast functor** from one user's action to all others. But the tricky part is network delays, etc., which go beyond pure category theory into categorical modeling of processes (using something like category of streams or a topos for time with partial orders). Without going too deep, category theory at least informs the architecture: one might structure group state as a **colimit** of individual states in a diagram shaped by the network topology (like each pair of users has a merging node etc.). The **universal property** of colimit would ensure that if there is any common state, it's merged, and differences are preserved otherwise – which is how collaborative data structures often work (CRDTs and such have a flavor of being colimits of edits). Implementation might not explicitly call it that, but by designing collaboration as a functor from the category of user-actions (with a partial order by time) to a global state, and ensuring it's left-adjoint to projection onto each user's view (meaning each user sees the most "equal or greater" state incorporating their actions), we get eventual consistency. Invariants like "everyone eventually sees the same thing" could thus be asserted as a limit property.

While the above can get complex, practically, one can embed simpler invariant checks: For instance, after a collaborative sequence, verify that all users have the same count of certain objects (like if one sees 3 overlays active, none should see 4, etc.). That's a simple equalizer check on states. If not equal, flash a warning or auto-correct by resending state. These invariants are effectively **kernel** conditions (differences should map to zero). Category theory tells us these conditions should be preserved by system operations; if an operation breaks an invariant, the design should be adjusted (maybe that operation belongs in a larger category where the invariant isn't fundamental).

Finally, consider **safety** in terms of forbidden states – category theory can define a subcategory of "safe states" and a functor that includes only those. Ensuring safety means the actual state functor factors through the inclusion of safe states. For example, if "neurostim: opt-in only, safety-clamped" is a requirement for BCI [46], that implies the system should never enter a state where neurostimulation is active without explicit opt-in flag. One can maintain a boolean in state that must be true for neurostim to occur. The safety subcategory is where either neurostim=off or (neurostim=on and opted_in=true). The functor representing state transitions must never map an initial state with opt_out to a final state with neurostim active (if so, that transition doesn't exist in the safe subcategory). We could have the type system or an invariant monitor ensure that (like a runtime assertion or a static type preventing calling a "activateStim()" method unless opt_in is true). The concept of **precondition** in programming (here opt_in) can be treated categorically as a **slice category** or **comma category** that forces a morphism to exist only when a certain arrow from opt_in object is present. This is technical, but it basically means building the UI logic so that any attempt to activate that feature goes first through a check/arrow from the "OptIn" object; if that arrow isn't there, the composition can't form, and thus the event is blocked (like a type error).

In summary, **state consistency functors** and invariants provide a formal backbone to ensure all parts of DonutOS remain in logical harmony as it grows in complexity. By designing or annotating the system with these functors (Model→View, UserIntention→Action, MultiUser→Global, etc.) and invariants (safety conditions, ambiguity handling, equality conditions), we can *prove or automatically verify* that certain unwanted situations cannot happen or will be caught. This goes a step beyond traditional testing – it's like baking the laws of the system directly into its architecture, so that many errors become *categorically impossible*. And if contradictions do occur, they manifest as non-commutative diagrams or functor failures, which the system can detect (like an assertion failing) and handle gracefully (perhaps by fallback logic).

5. Visual Proofs via Commutative Diagrams: UI for Trust and Understanding

One of the most compelling (and perhaps exotic) ideas is using **visual proofs** in the UI to explain and assure the user of correctness. In formal terms, a *commutative diagram* in category theory is essentially a picture that demonstrates that different paths of transformations yield the same result [19]. In DonutOS, many desirable properties can be expressed as "doing X then Y is same as doing Z" – which is a commutative diagram with two paths ($X \circ Y = Z$). Instead of hiding these facts deep in code, DonutOS could expose some of them as part of the user experience, enhancing transparency.

For example, consider the **overlay grouping** feature we imagined (pushouts merging overlays). After merging, the system could present a little diagram icon. Clicking it might show a simple triangle or square diagram illustrating that "Parameter P from Module A = Parameter P from Module B (unified)" by showing arrows from a source to both A and B that now converge. This is a proof that the modules are in sync. A more advanced scenario: if there are two different ways to reach a certain configuration (maybe the user can toggle an overlay on manually or it comes on automatically in a scene), the UI could reassure the user

that either way, it's the same end state. For instance, next to the overlay's toggle might be an info icon: "This overlay also activates when scene X is enabled." If the user clicks, it could show a diagram: *Scene X on* → *Overlay on* (via automation), and *User toggles overlay on* directly both lead to the same "Overlay active" state, with a note "Commutes: Scene X implies overlay, but you can also toggle it – effect is identical." This communicates that there's no conflict; if they toggle it off in scene X, maybe scene X will respect that or scene X is no longer fully active. If there was a potential conflict, that diagram might not commute (and the UI should have a policy to resolve it, which could also be depicted).

In safety-critical interactions, visual proofs can enhance **trust**. Suppose DonutOS integrates a *predictive model* that auto-adjusts something (like brightness based on ambient light). A skeptical user might wonder, "if I manually adjust brightness, will the auto-adjust fight me?" This scenario is about commutativity of manual override and auto-adjust algorithm. Ideally, when the user changes brightness, the system temporarily disables auto or shifts its baseline to avoid conflict – which is a design choice ensuring that the two paths (user sets level vs auto sets level) don't clash. The UI could show a small two-arrow loop icon meaning "auto-adjust is following your lead now" – essentially asserting a commutative result: (Past auto → current state) = (manual input → current state). If it didn't, the user would see oscillation, which means the diagram "Manual → State ← Auto" is not commutative (state differs if auto vs manual). Recognizing such a situation, the UI can visually break the link (maybe show a broken arrow if they truly conflict, indicating "system cannot maintain auto-adjust after your manual change unless re-enabled"). This is a way of making system *logic visible*.

At a deeper level, one could conceive of a **"Proof Mode"** for the truly curious or for developer-users. In proof mode, certain invariants or relationships in the system are visualized as diagrams that either appear with a green check (commutative – all good) or a warning symbol if not. Perhaps an overlay that monitors the **Edge-of-Chaos** band (CTI chaos band in spec [40]) might display a lattice or bifurcation diagram; if user's activity stays within a commutative square region (meaning stable regime), it shows a stable icon. If they push inputs into a chaotic regime (commutative diagrams break, meaning tiny changes lead to disproportionate results somewhere), the interface could animate some diagram lines diverging, signaling unstable dynamics. This is using the *metaphor* of commutative vs non-commutative as stable vs unstable, which mathematically can correlate with whether certain diagrams commute approximately or not (like nearly commuting diagrams might represent robust systems, non-commuting dramatic differences represent chaotic response).

One tangible simpler use: **tutorials and explanations**. If a user questions "why did action X happen now?", the system could answer by showing a little causal diagram: e.g., "because A → B and B → C, and you did A, thus C happened." This can be drawn as A→B→C highlight, with an equals sign to A→C path label "thus A led to C." That *is* a commutative triangle explanation, essentially the rule of composition in category theory made human-friendly. Many help systems do this in text ("Because you turned on filter A, results were sorted by date"), but a diagram can be clearer especially for visual thinkers. Given DonutOS's emphasis on holographic, visual resonance, providing explanatory diagrams resonates with its theme of making complex relations perceivable at a glance.

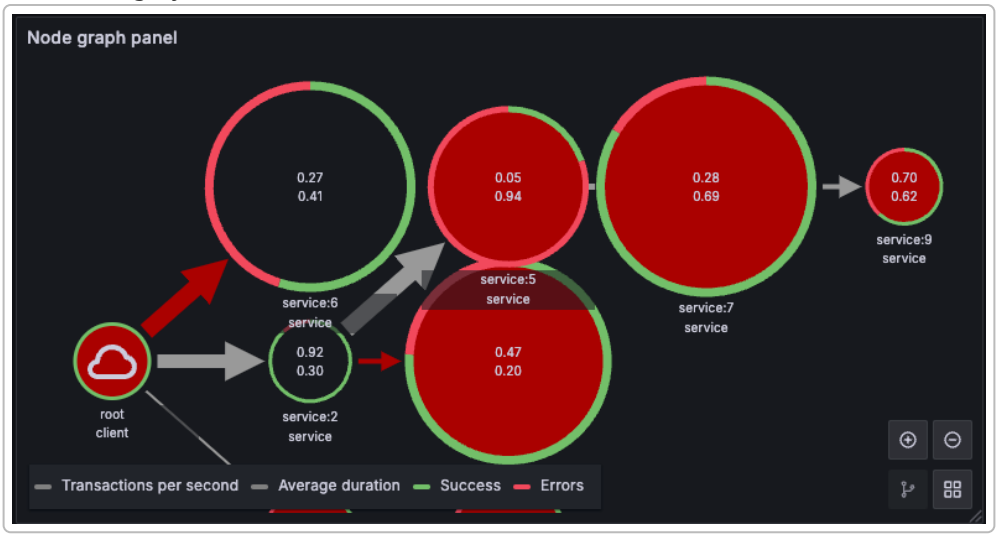
Finally, from a **mythic lens**, visual proofs echo ancient sacred diagrams – geometry demonstrating truths. One could whimsically relate commutative diagrams to the **Sigils** or **Mandala** of the system's inner workings – the user sees not just a bunch of data but the *relationships* as elegant geometry. For the interested, this fosters a deeper connection: the UI isn't magic, it's mathematics, and the user can witness

that structure. This might inspire confidence and mastery; they might start predicting outcomes because they internalize the diagrammatic logic.

To ensure practicality: not every user wants to see diagrams. So these can be contextual, appear on request or when confusion might arise. The key is the system *knows about its own relational structure*, so it can present it. Category theory makes that knowledge explicit in the design (with morphisms, constraints). Contrasting with a typical system where relationships are buried in code, here they're first-class, so representing them visually is straightforward. For example, we can easily fetch what dependencies a given overlay has because in our architecture it's literally a functor composition path. We can render that as arrows connecting components (which ties back to the node graph idea – indeed the node graph is one big commutative diagram depiction; visual proofs are just smaller focused pieces of it with specific semantic meaning).

In essence, **Visual Proofs via Diagrams** is about turning the often invisible "glue" of UI logic into visible, reassure artifacts. It aligns with DonutOS's ethos of explicitness (show locks, show ambiguity [47]) by also showing consistency and rationale. It's a novel UI paradigm where the system doesn't just do things, but can *explain* them with mini-diagrams – effectively teaching category theory intuition to the user unwittingly. Over time, the user might come to reason in these terms ("Ah, these two operations commute, so I can do them in any order"). That is empowerment: the user navigates the "torus of attention" with a clear mental map of cause and effect, symmetry and consequence, essentially seeing the Platonic form of their interactions.

Conclusion. In this exploratory paper, we have articulated how rigorous category theory can form the foundation of a speculative yet plausible future for DonutOS. By introducing formal categorical structures – objects, morphisms, functors, monoidal products, pushouts, and commutative diagrams – into the design of DonutOS's UI, we can achieve a system that is *composable*, *consistent*, and *transparent*. We analyzed current features like composable membranes, reactive overlays, and MVC-style interactions and showed they naturally align with categorical concepts [1] [14] [2]. Building on that, we proposed future directions: a **Scene Algebra** to combine and blend modes (so the UI becomes an algebraic playground of states), **Node-Graph Overlays** to let users visually program and rewire their interface (empowering them to leverage the system's internal category)



, **Pushout gluing tools** for seamlessly integrating new modules (ensuring that when two features meet, they join coherently with nothing duplicated and nothing lost) [30] [42], **State Consistency Functors** to enforce invariants across an increasingly rich state space (so that as complexity grows, fundamental behaviors remain correct and safe), and **Visual Proofs** in the interface to communicate the system's workings and reassure or educate the user (turning abstract correctness into tangible diagrams they can see and trust) [19].

Throughout, we wove in DonutOS's metaphors – the torus, fractal-holographic attention, levogyre shells, Platonic overlays – not as mere ornament but as intuitions for these formal ideas. The torus itself is a shape rich in symmetry (a product of circles, each a category of rotations), and our formalism reflected that in treating rotations as automorphisms and phase locks as commutative diagrams of rotations. The **Platonic solids overlays** hint at underlying group theory, which ties to monoidal and symmetric categories in the UI. The **levogyre shells** inspired the idea of layered cyclic structures and locking (which we captured through commutative diagrams enforcing synchronization). Even the ethos of *"weak-control, exploratory but safe"* we interpreted through category theory: giving the user freedom (composable operations) but within a structured space where certain bad compositions are simply not arrows in the category (thus prevented), and ambiguity is not collapsed (multi-valued functors are exposed to the user rather than forcing a single value). In a sense, category theory is the "Platonic form" behind DonutOS's experiential form – it is the crisp blueprint of relationships that gives rise to the rich, dynamic interface (the shadows on the cave wall, to extend Plato's allegory, but we now have a way to reason about the objects casting them).

Realizing this vision in full would be a significant engineering endeavor – requiring merging HCI design with formal methods – but we already see trends in that direction (for example, the increasing use of **type systems** in UI frameworks to catch errors, or the use of **reactive declarative languages** that abstract the when and how of updates). DonutOS could become a pioneering example of a * formally grounded* interface: one that is not only user-friendly and imaginative but also mathematically "correct by construction." The benefit would be a system that can scale in complexity without breaking down, that can integrate novel inputs (brain signals, group data) gracefully, and that can offer users unprecedented control to tailor the system to their needs without fear – because the algebra of the interface prevents inconsistent or unsafe configurations.

In closing, by marrying the **rigor of category theory** with the **speculative vision of DonutOS**, we chart a path toward interfaces that are **modular, reliable, and transparent** in ways previously thought too theoretical. As computing interfaces expand from 2D screens to AR/VR spaces, to neural interfaces and beyond, such a solid theoretical foundation might be what keeps the *fractal torus of attention* coherent across scales – personal to planetary – fulfilling DonutOS's goal of a toroidal, holographic field of attention that remains harmoniously composed [1]. In a world of increasing complexity, this approach offers *coherence over control* [45]: empowering users with a system that is deeply **comprehensible** and **controllable** because its very design guarantees consistency and reveals its logic. DonutOS's future, through the lens of category theory, looks not only exciting and feature-rich but grounded in a timeless mathematical truth: *the whole is greater than the sum of its parts* when those parts are linked by the right arrows in the right category.

Sources:

- DonutOS Project Specification (Snapshot) – core features, extensions, and principles [48] [29].

- Milewski, B. *“Category: The Essence of Composition.”* (composition with functors/monads in software) [3].
- Panchekha, P. *“Categories for Reactive Programming.”* (category-theoretic description of FRP) [14].
- Haskell For All – *“Model-View-Controller, Haskell-style.”* (MVC with functors and monoids) [21].
- Wikipedia – *“Pushout (Category Theory).”* (gluing objects via universal construction) [41] [30].
- Wikipedia – *“Commutative Diagram.”* (paths with same start and end yield same result) [19].
- Arthur Xavier, *“Comonads for UIs”* (thesis introduction on lack of formal UI models and need for composition) [2].

---

[1] [7] [10] [15] [25] [27] [28] [29] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [43] [44] [45] [46] [47] [48] DONUT_SPEC.md
file://file_000000002050720aa9e76987ae412901

[2] [23] [24] arthurxavierx.github.io
https://arthurxavierx.github.io/ComonadsForUIs.pdf

[3] The Elegance Principle: Why Mathematicians Make Good Software Architects - SoftwareSeni
https://www.softwareseni.com/the-elegance-principle-why-mathematicians-make-good-software-architects/

[4] [5] [6] [17] Categories from scratch · Raphael Poss · workbench
https://dr-knz.net/categories-from-scratch.html

[8] [PDF] Category Theory for Software Modeling and Design
https://angelineaguinaldo.com/assets/slides/HCAM_Seminar___Oct_29_2020.pdf

[9] [PDF] Monoidal Category Theory Unifying Concepts in Mathematics ...
http://www.sci.brooklyn.cuny.edu/~noson/MCTslides06.pdf

[11] Functor - Wikipedia
https://en.wikipedia.org/wiki/Functor

[12] [13] [26] Section 7.2 (00V1): Presheaves—The Stacks project
https://stacks.math.columbia.edu/tag/00V1

[14] [16] Categories for Reactive Programming
https://pavpanchekha.com/blog/frp-categories.html

[18] How are functors in Haskell and OCaml similar? - Stack Overflow
https://stackoverflow.com/questions/16353066/how-are-functors-in-haskell-and-ocaml-similar

[19] Commutative diagram - Wikipedia
https://en.wikipedia.org/wiki/Commutative_diagram

[20] [21] [22] Haskell for all: Model-view-controller, Haskell-style
https://www.haskellforall.com/2014/04/model-view-controller-haskell-style.html

[30] [42] category theory - Applications of Pushouts - Mathematics Stack Exchange
https://math.stackexchange.com/questions/2470031/applications-of-pushouts

[41] Pushout (category theory) - Wikipedia
https://en.wikipedia.org/wiki/Pushout_(category_theory)