CE2 Architecture Overview

Code Explorer 2 (CE2) is a multi-level visualization app for exploring PureScript package ecosystems. It demonstrates patterns for building complex, stateful visualization applications with Halogen and PSD3.

Core Architecture

```
AppShell (data loading, error handling)
    │
    ▼
SceneCoordinator (state machine, navigation, transitions)
    │
    ├── GalaxyBeeswarmViz (Halogen component wrapping D3 simulation)
    ├── BubblePackBeeswarmViz (nested circle packing + force simulation)
    ├── CirclePackViz (three-layer neighborhood layout)
    ├── ModuleTreemapViz (treemap with click-to-panel)
    └── SlideOutPanel (documentation/source viewer)
```

Key Patterns

1. Scene-Based State Machine

Navigation is modeled as a state machine with explicit scenes:

```
data Scene
  = GalaxyTreemap           -- Registry overview (568 packages)
  | GalaxyBeeswarm          -- Topo-ordered beeswarm
  | SolarSwarm              -- Project packages with modules inside
  | PkgNeighborhood String  -- deps | focal | dependents
  | PkgTreemap String       -- Module treemap
  | PkgModuleBeeswarm String -- Module flow overlay
```

Benefits: Clear mental model, predictable transitions, easy to add new views.

2. Derived Over Stored State

State that can be computed is derived, not stored:

```
-- Theme is derived from scene, not stored
themeForScene :: Scene -> ViewTheme
themeForScene GalaxyTreemap = BlueprintTheme
themeForScene SolarSwarm = BeigeTheme
themeForScene (PkgTreemap _) = PaperwhiteTheme
```

```
-- In render:
let theme = themeForScene state.scene
```

Benefits: No desync between stored and computed values, simpler state.

3. Coordinator Owns Navigation State

The coordinator tracks all navigation-relevant state. Child components are stateless with respect to navigation:

```
-- Coordinator state
type State =
  { scene :: Scene
  , previousScene :: Maybe Scene
  , scope :: BeeswarmScope
  , viewMode :: ViewMode
  , panelOpen :: Boolean
  , panelContent :: PanelContent
  -- ... data from parent
  }
```

4. Child Components: lastInput Pattern

Child visualization components store only internal state (simulation handles, listeners) plus `lastInput` for change detection:

```
-- Child component state (GalaxyBeeswarmViz, BubblePackBeeswarmViz, etc.)
type State =
  { handle :: Maybe SimulationHandle  -- Internal: D3 simulation
  , initialized :: Boolean             -- Internal: first render done
  , actionListener :: Maybe Listener   -- Internal: D3 → Halogen bridge
  , lastInput :: Input                 -- For change detection only
  }

handleAction = case _ of
  Receive input -> do
    state <- H.get
    let packagesChanged = input.packages /= state.lastInput.packages
    H.modify_ _ { lastInput = input }
    when packagesChanged $ startVisualization input  -- Use input directly
```

Benefits: No state desync between parent Input and child State.

5. D3 → Halogen Event Bridge

D3 events (clicks, hovers) are bridged to Halogen via subscriptions:

```
-- In Initialize:
{ emitter, listener } <- liftEffect HS.create
void $ H.subscribe emitter
H.modify_ _ { actionListener = Just listener }

-- D3 callback (passed to render function):
onPackageClick: \name -> HS.notify listener (HandlePackageClick name)

-- Action handler:
HandlePackageClick name -> H.raise (PackageClicked name)
```

6. Orthogonal State Dimensions

State dimensions are independent axes, not combined into a single ADT:

```
Scene     × Scope           × ViewMode      × Panel
(where)    (what's visible)  (how shown)    (side content)

GalaxyBeeswarm × ProjectOnly × PrimaryView × closed
SolarSwarm     × Transitive  × ChordView   × open(module)
```

Benefits: Combinatorial flexibility without explosion of scene variants.

7. GUP for Scope Filtering

Scope changes use D3's General Update Pattern (enter/update/exit) for smooth filtering:

```
SetScope targetScope -> do
  H.modify_ _ { scope = targetScope }
  -- Child component detects scope change via lastInput comparison
  -- Calls setScope on simulation handle for animated enter/exit
```

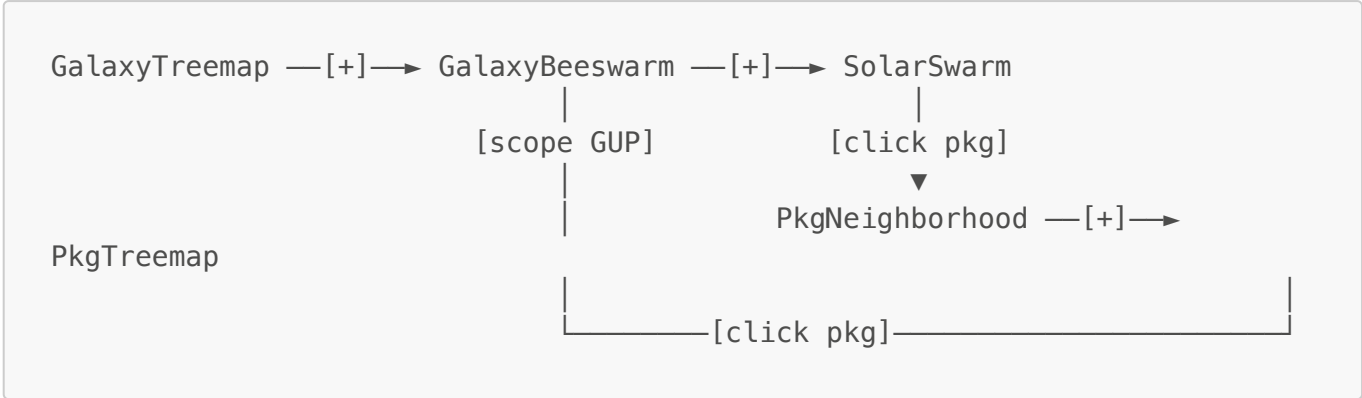
8. Position Capture for Transitions

Animated transitions capture positions from source visualization:

```
NavigateForward -> case state.scene of
  GalaxyTreemap -> do
    positions <- liftEffect $ getCellPositions "#container"
    H.modify_ _ { capturedPositions = Just positions }
    handleAction (NavigateTo GalaxyBeeswarm)
```

Child components receive `initialPositions` in Input and spawn nodes at those positions before starting simulation.

State Machine Transitions



- **+**: Explicit navigation (captures positions, animated transition)
- **scope GUP**: Filter in place, no navigation
- **click**: Navigate + open panel

File Structure

```
ce2-website/src/
├── Main.purs                    # Entry point
├── Scene.purs                   # Scene ADT, parentScene, sceneLabel
├── Types.purs                   # ViewTheme, ColorMode, BeeswarmScope
├── Containers.purs              # DOM container IDs/selectors
├── Data/
│   ├── Loader.purs              # API calls, data types
│   └── Filter.purs              # Scope filtering logic
├── Component/
│   ├── AppShell.purs            # Top-level: data loading
│   ├── SceneCoordinator.purs    # State machine, navigation
│   ├── GalaxyBeeswarmViz.purs   # Package beeswarm
│   ├── BubblePackBeeswarmViz.purs # Packages with modules inside
│   ├── CirclePackViz.purs       # Three-layer neighborhood
│   ├── ModuleTreemapViz.purs    # Module treemap
│   └── SlideOutPanel.purs       # Documentation panel
└── Viz/
    ├── PackageSetBeeswarm.purs  # D3 simulation (FFI)
    ├── PackageSetTreemap.purs   # D3 treemap (FFI)
    ├── BubblePackBeeswarm.purs  # Nested pack + simulation
    ├── ModuleTreemap.purs       # Module-level treemap
    └── ScaleTransition.purs     # Position capture utilities
```

Design Decisions

1. **Why scenes, not routes?** CE2 is a visualization explorer, not a traditional web app. Scene-based navigation maps naturally to "zoom levels" in a Powers of Ten metaphor.

2. **Why coordinator pattern?** Centralizes state machine logic. Child components become simpler (just render what they're told).

3. **Why lastInput instead of copying?** Prevents desync. The source of truth for "what should be displayed" is always the parent's Input.

4. **Why explicit scope transitions?** Auto-escalation (scope change triggering navigation) was confusing. Users now understand: scope = filter, + = navigate.

5. **Why orthogonal dimensions?** Avoids scene explosion. 6 scenes × 4 scopes × 3 view modes = 72 combinations, but only ~15 lines of state.

Future Considerations

- **URL routing**: Could map scene + scope + viewMode to URL for shareable links
- **Undo/redo**: previousScene could become a full history stack
- **Persistence**: Panel state and scope preferences could persist to localStorage
- **Semantic zoom**: Click anywhere to drill down, back to zoom out (ZUI paradigm)