Stride’s **Graphics Compositor's dependency graph UI** in Stride is backed by a robust **internal graph framework**, and yes, it can be repurposed for general dependency visualization or graph-based tools.

**✅ What It Uses Internally**

Stride’s **Graphics Compositor** and asset system use:

* A **dataflow graph framework** (node → slot → connection)
* Backed by **YAML-serialized assets**
* Visualized in Game Studio via the **GraphView** control in their editor tooling

This is implemented via:

* GraphView, NodeViewModel, SlotViewModel, etc.
* Built on **Stride.Core.Presentation** (WPF-based MVVM UI framework)
* Rendered using **Avalonia** or WPF, depending on host editor

**🧩 Options to Reuse It**

1. **Reference Stride’s Editor UI Libraries**
   * Found in Stride.Assets.Presentation, Stride.Core.Presentation
   * You can extract and reuse GraphView and its MVVM backbone
2. **Use the concepts only**
   * Implement a simplified version using **Avalonia** or **ImGui** in your own tools
   * Maintain the data structures (nodes, connections, update propagation)
3. **Watch for future tooling**
   * Stride devs have discussed separating the **GraphView system** for broader use

**💡 Use Case in MotionNotions**

You could use this tech to:

* Show agent routing or motion graphs
* Build an **event timeline editor**
* Visualize **connectors between PolyShapes** as a flow map
* Offer **material, animation, or behavior graph editors**

**⚠️ Licensing Note**

Stride is under the **MIT license**, so you’re free to extract and adapt that tech — just mind that editor-side components are built for WPF, not runtime UI.

Would you like a sample that mimics GraphView using Avalonia or Stride UI primitives?

Excellent, sir — **Avalonia** is a solid choice for building your own editor-grade **graph UI** similar to Stride’s Compositor. It’s fast, open, and cross-platform, and unlike Stride’s WPF-based tooling, it plays nicely with Linux and modern UIs.

**✅ What You’d Be Building**

A minimal **GraphView-like** control in Avalonia could include:

* GraphCanvas: the control surface
* GraphNodeViewModel: represents each node
* GraphSlotViewModel: input/output slots
* ConnectionView: renders connections (lines or curves)
* GraphEditor: handles mouse events and drag+drop wiring

**🔹 Technologies**

* **Avalonia 11+**
* **MVVM pattern** (with ReactiveUI or CommunityToolkit)
* Canvas or Panel for rendering
* Path/Line for connections

**💡 Suggested Structure**

GraphEditor (UserControl)

└── Canvas or Panel

├── GraphNodeControl (Positioned by Canvas.Left/Top)

│ ├── Title

│ └── SlotControls (input/output)

└── ConnectionPath (Bezier or straight line)

**🧩 Integrates Nicely With MotionNotions**

* Design routes visually with connectors
* View dependency chains (e.g. agent → path → behavior)
* Show live animation trees or scripted event chains

Would you like a working sample with:

* A canvas
* Draggable nodes
* Click-to-connect input/output slots
* Using Avalonia 11 + PathGeometry?

Your `MotionNotions.Graphs` core is ready, sir — a clean, Avalonia-compatible foundation:

\* `GraphNode` with input/output slots

\* `GraphConnection` enforcing Output → Input logic

\* `GraphModel` tracks nodes and connections, supports querying

Next: shall I scaffold an Avalonia UI that renders this model visually, with draggable nodes and connection lines?

6 June 2025 (Friday)

Let's review what we are doing. We want a generalized dependency diagram that allows us to display and change node dependencies in Avalonia. We are going to want connection points on the nodes and can be used to draw these connection arrows. I prefer left to right flow, so Input connections (anchors) are on the left and top and Output anchors will be on the right and bottom. The UI will allow the display and building of these connections. We will use the microsoft XML standard for reading/writing the dependency graphs (I don't remember its name/acronym) MSAGL? or something like that. The connection lines (beziers) will "steer around" the node boxes ultimately.

Perfect summary, sir. Here's a clear breakdown of your design goals and the components we'll focus on:

**🎯 Primary Objective**

A **generalized dependency graph editor** in Avalonia that:

* Displays and edits **node-based dependency diagrams**
* Supports **bidirectional interaction** (view and modify)
* Uses **MS-standard XML** format (likely **DGML** – *Directed Graph Markup Language*) for serialization

**🧱 Architecture Overview**

**✅ Backend (DependencyGraphCore)**

* GraphNode – Holds Id, Name, Inputs, Outputs
* GraphSlot – Defines Input or Output slots
* GraphConnection – Links one node's output to another node’s input
* GraphModel – Manages all nodes and connections

**✅ ViewModel Layer (DependencyGraphEditor.ViewModels)**

* GraphNodeViewModel – Adds position info (X, Y) to GraphNode
* MainViewModel – Holds all nodes (and eventually connections)

**✅ UI Layer (DependencyGraphEditor.Views)**

* GraphEditorView.axaml – Displays:
  + Each node (as a draggable visual with connection anchors)
  + All connections (as steerable Bezier curves)
* Interaction support for:
  + Dragging nodes
  + Clicking anchors to draw connections
  + Visual feedback during connection

**🔗 Connections**

* **Anchor Placement**:
  + Input: **Left** and **Top**
  + Output: **Right** and **Bottom**
* **Visuals**:
  + Use **Bezier curves** for smooth links
  + Will need layout logic to **route around nodes**

**📄 Data Format**

* **DGML (Directed Graph Markup Language)** is likely the standard you referred to.
  + Used by Visual Studio
  + XML-based
  + Represents nodes, links, and categories

**✅ Next Steps**

Would you like to proceed with:

1. Implementing **anchor visuals** on the nodes (left/top = input, right/bottom = output)?
2. Supporting **interactive connection drawing** (click anchor → drag → connect)?
3. Integrating **DGML I/O** for saving/loading graphs?

Let me know which you'd prefer to focus on next.

## Step 1 - Anchors

Excellent, sir. For **Step 1: Anchor Visuals**, we’ll render **connection points (anchors)** on each GraphNodeViewModel as follows:

**🧲 Anchor Rules**

* **Left** & **Top**: Input slots
* **Right** & **Bottom**: Output slots
* Each slot becomes a small Ellipse on the node border

**✅ Update to DrawNode**

We'll modify DrawNode(GraphNodeViewModel gnvm) to:

* Add four anchor Ellipse elements at the cardinal positions
* Bind tooltips or tags to indicate slot name/type (for future interaction)

**🔧 Code Snippet (partial)**

Add inside DrawNode, after creating the node Border:

foreach (var slot in gnvm.Node.Inputs)

{

var leftAnchor = new Ellipse

{

Width = 8,

Height = 8,

Fill = Brushes.LightBlue,

Stroke = Brushes.DarkBlue,

StrokeThickness = 1,

[Canvas.LeftProperty] = -4, // half width outside

[Canvas.TopProperty] = border.Height / 2 - 4,

Tag = slot

};

Canvas.SetZIndex(leftAnchor, 2);

nodeContainer.Children.Add(leftAnchor);

}

// Repeat similar for Outputs: place on Right/Bottom

**📍 Next Actions**

1. I’ll give you the **full updated DrawNode() method** with anchor placement
2. We’ll ensure anchors are distinct and responsive
3. Then we’ll handle pointer events in step 2

Shall I continue with the full anchor-rendering version of DrawNode, sir?